VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PREDIKTIVNÍ SYNTAKTICKÁ ANALÝZA S HLUBOKÝMI ZÁSOBNÍKY

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JIŘÍ VIKTORIN
AUTHOR

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

PREDIKTIVNÍ SYNTAKTICKÁ ANALÝZA S HLUBOKÝMI ZÁSOBNÍKY

PREDICTIV SYNTACTIC ANALYSIS WITH DEEP PUSHDOWN

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Jiří Viktorin
AUTHOR

VEDOUCÍ PRÁCE prof. RNDr. Alexander Meduna, CSc.
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Viktorin Jiří**
Obor: Informační technologie
Téma: **Prediktivní syntaktická analýza s hlubokými zásobníky**
Kategorie: Překladače

Pokyny:

1. Dle pokynů vedoucího se seznamte detailně s automaty s hlubokými zásobníky a jejich vlastnostmi.
2. Dle pokynů vedoucího studujte nové vlastnostmi těchto automatů.
3. Navrhněte metodu obecné prediktivní syntaktické analýzy, která je založena na automaty s hlubokými zásobníky.
4. Studujte užití metody syntaktické analýzy navržené v předchozích bodech. Zaměřte se na překladače programovacích jazyků. Navrhněte vhodný programovací jazyk a sestrojte jeho syntaktický analyzátor, který provádí syntaktickou analýzu na základě této metody. Testujte výsledný syntaktický analyzátor.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

1. Meduna, A.: Automata and Languages, Springer, London, 2000

Při obhajobě semestrální části projektu je požadováno:

1. Body 1 - 3.

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

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

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

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Jiří Viktorin**
Id studenta: 84109
Bytem: Fr.Formana 238/33, 700 30 Ostrava
Narozen: 28. 07. 1985, Vyškov
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP: Prediktivní syntaktická analýza s hlubokými zásobníky
Vedoucí/školitel VŠKP: Meduna Alexander, prof. RNDr., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☒ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Abstrakt

V této práci se zaměřím na implementaci hlubokých zásobníkových automatů, které jsou generalizací klasických zásobníkových automatů. Tyto automaty mají větší sílu, ale nemají sílu na rozpoznávání všech kontextových gramatik. Tato síla je dána díky tomu, že mohou expandovat neterminální symbol i v hloubce větší než 1.

Klíčová slova

Zásobníkový automat, hluboký zásobníkový automat, Gramatika, Chomského klasifikace gramatik, formální jazyky

Abstract

This paper introduces a generalization of classical pushdown automata-deep pushdown automata. Deep pushdown automata expand a pushdown symbol in a depth defined by a rule. Deep pushdown expands a non-terminal in the deep defined in the rule and this non-terminal need not be on the top of pushdown.

Keywords

Pushdown automaton, deep pushdown automaton, grammar, formal languages, Chomsky grammar definition

Citace

Jiří Viktorin, Prediktivní syntaktická analýza s hlubokými zásobníky, bakalářská práce, Brno, FIT VUT v Brně, 2007

Prediktivní syntaktická analýza s hlubokými zásobníky

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením
prof. RNDr. Alexandra Meduny, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jiří Viktorin
15. 5. 2007

Poděkování

Zde bych chtěl poděkovat prof. Alexandru Medunovi, za odbornou poradu při konzultacích a za doporučení při řešení implementace automatu. Chtěl bych také poděkovat kolegovi Peteru Solárovi za pomoc při implementaci a řešení otázek efektivity kódu.

Obsah

Úvod

V této práci se soustřeďuji na hluboké zásobníkové automaty a jejich praktickou realizaci ve výpočetním prostředí. Tyto automaty definoval prof. Alexander Meduna ve svém článku „Deep pushdown automata“. Hluboké zásobníkové automaty mohou být používány například při syntaktické analýze v překladačích, nebo se také mohou používat při kontrole vstupních řetězců a jejich náležitosti ke gramatice, která je dána pomocí pravidel. Tato pravidla můžeme přizpůsobit automatu a ten podle nich určí, zda vstupní řetězec bude akceptovat či nikoli. Hluboké zásobníkové automaty mají větší výpočetní sílu než klasické zásobníkové automaty. Je toho dosaženo tím, že mohou provést rozvoj i v jiné hloubce zásobníku, než jen na jeho vrcholu. Gramatiky, které akceptují, náleží do skupiny mezi kontextovými a bezkontextovými. Jestliže bychom použili automat o maximální hloubce ∞ dostaneme sílu srovnatelnou s Turingovým strojem.

1 Gramatiky

1.1 Definice gramatik

Pojem gramatika úzce souvisí s problémem reprezentace jazyka. Způsob vymezení pravidel jazyka výčtem všech vět tohoto jazyka je nepoužitelný, a to zejména pro jazyky nekonečné, ale i pro obsáhlé jazyky konečné. Také matematické vyjádření je použitelné pouze pro jazyky s jednoduchou strukturou.

Základní požadavek kladený na gramatiku je konečnost její reprezentace. Používá se dvou konečných disjunktních abeced.

1) Množina nonterminálních symbolů N, tzv. nonterminálů
2) Množina terminálních symbolů ∑, tzv. terminálů

Nonterminální symboly jsou v roli pomocných symbolů, které se používají ke znázornění syntaktických celků. Terminální symboly jsou shodné s abecedou, nad kterou je definován jazyk. Sjednocením těchto množin získáme tzv. slovník gramatiky, což je výčet všech symbolů, které mohou být v gramatice použity.

Pro gramatiku identifikátorů může mít slovník například následující podobu:

N = {<identifikátor>, <písmeno>, <číslice>}

∑ = {A-Z, a-z, 0-9}

Z toho získáváme, slovník, který bude mít 65 symbolů, jež se ve větách jazyka s touto gramatikou mohou vyskytovat.

Gramatika představuje tzv. generativní systém, což znamená, že aplikací tzv. přepisovacích pravidel můžeme z jednoho výchozího nonterminálního symbolu generovat řetězce, které jsou tvořeny pouze terminálními symboly. Základem gramatiky je tedy konečná množina přepisovacích pravidel *P*. Každé toto pravidlo má tvar uspořádané dvojice řetězců (α,β), a stanovuje substituci řetězce α za řetězec β. Řetězec α je podřetězcem generovaného jazyka a obsahuje minimálně jeden neterminální symbol.

Jestliže na řetězec *x* aplikujeme pravidlo (α,β), a dostaneme z něj řetězec *y*, říkáme, že jsme řetězec *y* odvodili, nebo také derivovali, pomocí pravidla (α,β) z řetězce *x*.

Gramatika je pak dána čtveřicí

$$G = (N, \Sigma, P, S)$$

N - konečná množina neterminálních symbolů

∑ - konečná množina terminálních symbolů, přičemž $N \cap \Sigma = \emptyset$

P - konečná množina pravidel

S - startující symbol, kde $S \in N$

Prvek (α,β) z množiny *P* nazýváme přepisovacím pravidlem a zpravidla jej zapisujeme ve tvaru $\alpha \rightarrow \beta$. Řetězec α pak nazýváme levou a řetězec β pravou stranou přepisovacího pravidla. Jestliže množina pravidel *P* obsahuje přepisovací pravidla tvaru $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \cdots, \alpha \rightarrow \beta_n$ můžeme tato přepisovací pravidla zapsat zkráceně $\alpha \rightarrow \beta_1 | \beta_2 | \cdots | \beta_n$

Nechť $G = (N, \Sigma, P, S)$ je gramatika a *x*, *y* jsou řetězce z $(N \cup \Sigma)^*$. Mezi řetězci *x*, *y* platí relace $\Rightarrow_G$ nazývaná jako přímá derivace, jestliže můžeme řetězce *x*, *y* vyjádřit ve tvaru

$$x = u\alpha v$$

$$y = u\beta v$$

kde *u*, *v* jsou libovolné řetězce z $(N \cup \Sigma)^*$ a existuje přepisovací pravidlo $\alpha \rightarrow \beta$ v množině *P*. Jestliže mezi řetězci *x*, *y* platí relace přímé derivace, pak toto můžeme zapsat jako $x \Rightarrow_G y$ a říkáme, že řetězec *y* lze přímo derivovat z řetězce *x* v gramatice G. Jestliže je z kontextu patrné, že se jedná o derivace v gramatice G, nemusíme toto pod znakem derivace ⇒ uvádět.

Ve stejné gramatice nechť platí, že mezi řetězci *x*, *y* existuje relace $\Rightarrow^+$, zvaná derivace, jestliže existuje posloupnost přímých derivací $v_{i-1} \Rightarrow v_i$ kde i = 1, …, *n* a $n \geq 1$ taková, že platí

$$x \Rightarrow v_0 \Rightarrow v_1 \Rightarrow \cdots \Rightarrow v_{n-1} \Rightarrow v_n \Rightarrow y$$

Platí-li v gramatice G pro řetězce *x*, *y* relace derivace nebo relace ekvivalence, jedná se o tranzitivní a reflexivní uzávěr relace přímé derivace. Ten zapisujeme jako $\Rightarrow^*$

Řetězec α nazýváme větnou formou, jestliže existuje tranzitivní a reflexivní uzávěr $S \Rightarrow^* \alpha$. To znamená, že řetězec α lze vygenerovat ze startujícího symbolu *S*. Větná forma obsahující jen terminální symboly se nazývá věta. Jazyk generovaný gramatikou G – L(G) je pak definován množinou všech vět

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

1.2 Chomského klasifikace gramatik

Důležitou oblast informatiky tvoří teorie formálních jazyků. Základy jí položil roku 1956 Noam Chomsky, americký matematik, který v souvislosti se studiem přirozených jazyků vytvořil matematický model gramatiky jazyka.

Původně se snažil formalizovat popis přirozeného jazyka, aby mohl být automatizován překlad z jednoho přirozeného jazyka do druhého, nebo aby člověk mohl komunikovat s počítačem pomocí přirozeného jazyka. To se však ukázalo jako velmi obtížné. Začala se vyvíjet vlastní teorie jazyků, která obsahuje výsledky v podobě matematicky dokázaných tvrzení – tzv. teorémů. Tato teorie pracuje se dvěma duálními entitami představujícími abstraktní matematické stroje. Jsou to gramatiky a automaty. Zatímco gramatika dokáže strukturu popisovat, automat tuto strukturu dokáže rozpoznat.

Chomského klasifikace gramatik a jazyků, nebo také jako hierarchie jazyků, vymezuje 4 typy gramatik. Rozděluje je podle tvaru přepisovacích pravidel, jež jsou obsaženy v množině přepisovacích pravidel *P*. Tyto gramatiky se označují jako typ 0, typ 1, typ 2 a typ 3.

1.2.1 Gramatiky typu 0

Gramatiky typu 0 obsahují pravidla nejobecnějšího tvaru, jaký byl popsán v definici gramatik. Tato pravidla nemají žádná omezení a z tohoto důvodu se nazývají gramatikami neomezenými.

Příklad:

$$G = (\{A, B\}, \{a, b\}, P, S)$$

$$P = \{ \quad A \rightarrow AbB|a$$

$$AbB \rightarrow baB|BAbB$$

$$B \rightarrow b|\varepsilon \ \}$$

1.2.2 Gramatiky typu 1

Gramatiky typu 1 mají pravidla ve tvaru

$$\alpha A\beta \rightarrow \alpha\gamma\beta;\ A \in N;\ \alpha, \beta \in (N \cup \Sigma)^*;\ \gamma \in (N \cup \Sigma)^+ \ nebo\ S \rightarrow \varepsilon$$

Gramatiky typu 1 se nazývají gramatikami kontextovými, neboť pravidla těchto gramatik implikují, že nonterminální symbol A může být nahrazen řetězcem γ pouze tehdy, je-li jeho pravým kontextem β a levým kontextem α.

Kontextové gramatiky neobsahují pravidla $\alpha A\beta \rightarrow \alpha\beta$, tj. nepřipouštějí, aby byl nonterminální symbol nahrazen prázdným řetězcem. Jedinou výjimkou je pravidlo $S \rightarrow \varepsilon$. V důsledku těchto pravidel nemůže v gramatikách typu 1 dojít při generování ke zkrácení řetězce, tudíž platí-li $\lambda \Rightarrow \mu$ pak $|\lambda| \leq |\mu|$ s výjimkou pravidla $S \rightarrow \varepsilon$.

1.2.3 Gramatiky typu 2

Gramatiky typu 2 obsahují pravidla ve tvaru:

$$A \rightarrow \gamma, A \in N, \gamma \in (N \cup \Sigma)^*$$

Gramatiky typu 2 se nazývají gramatikami bezkontextovými, neboť substituci levé strany pravidla γ za nonterminál A můžeme provést bez ohledu na kontext, ve kterém je nonterminál A uložen. Na rozdíl od kontextových gramatik bezkontextové gramatiky mohou obsahovat pravidla tvaru $A \rightarrow \varepsilon$. Každou bezkontextovou gramatiku lze transformovat, aby obsahovala nejvýše jedno přepisovací pravidlo s prázdným řetězcem na pravé straně, aniž by se změnil jazyk touto gramatikou generovaný. V takovém případě bezkontextové gramatiky, stejně jako v případě kontextové gramatiky, se symbol S nesmí objevit v žádné pravé straně přepisovacího pravidla gramatiky G.

1.2.4 Gramatiky typu 3

Gramatiky typu 3 obsahují pravidla ve tvaru

$$A \rightarrow xB|x \quad A, B \in N, x \in \Sigma^*$$

Gramatiky, které se generují těmito pravidly, se nazývají pravé lineární gramatiky. Tyto gramatiky mají jediný možný nonterminál pravé strany úplně vpravo. Gramatiky typu 3 se také nazývají regulárními gramatikami.

1.2.5 Jazyky generované gramatikami

Jazyk generovaný gramatikou typu *i*, *i* = 0, 1, 2, 3 se nazývá jazykem typu *i*. Podle názvů gramatik hovoříme také o jazycích neomezených, pro typ 0, kontextových, pro typ 1, bezkontextových, pro typ 2, nebo regulárních pro typ 3.

2 Definice automatů

Automaty jsou výpočetní stroje, které jednají podle pravidel jim určeným. Automaty bez zásobníku pracují tak, že podle znaku, který načtou ze vstupního řetězce, přejdou do stavu, který je definován v pravidlech přechodů. Tyto automaty nemohou kontrolovat pravidla, ve kterých se mají kontrolovat párové znaky, jako jsou například závorky, nebo v programování „begin“ a „end“.

K tomu se používají právě klasické zásobníkové automaty, které využívají zásobníku ke kontrole složitějších soustav. Nemohou však stále kontrolovat všechny řetězce, jelikož nemají stále dostatečnou sílu. U těchto automatů se využívá zásobník, do kterého se postupně vkládá dočasný řetězec, který se poté porovnává se vstupním řetězcem. Tyto automaty kromě vstupní abecedy využívají ještě tzv. zásobníkovou abecedu, která je oproti vstupní abecedě rozšířena o symboly tzv. neterminální, které zásobníkový automat může rozvinout na řetězec dalších neterminálních symbolů, vstupních symbolů, nebo jejich libovolnou kombinaci. Tyto neterminální symboly však může klasický zásobníkový automat rozvinout pouze, pokud se tento nachází na vrcholu zásobníku. Pokud se na vrcholu zásobníku nachází stejný symbol, jako na vstupu, je tento symbol vyjmut ze zásobníku a čtecí hlava se posune na další symbol vstupu. Pokud se symbol na vrcholu zásobníku neshoduje se symbolem na začátku vstupního řetězce, je tento řetězec označen jako neakceptovaný. Pokud automat přejde až do stavu, který je označen jako koncový, je přečten celý vstupní řetězec a přitom je zásobník vyprázdněn, je řetězec označen za akceptovaný – přijímaný daným automatem.

Pokud potřebujeme zajistit vyšší výpočetní výkon automatu, můžeme jej implementovat jako hluboký zásobníkový. Tento automat oproti klasickému zásobníkovému může rozvinout nejen neterminální symbol na vrcholu zásobníku, ale i ten, který se nachází ve větší hloubce zásobníku. Pokud potřebujeme tento automat implementovat, musíme umožnit automatu procházet zásobník i v jiné hloubce, než je hloubka 1 – vrchol zásobníku. Musíme implementovat i to, aby v této hloubce mohl automat řetězec rozvinout. Pokud tento automat implementujeme, dostaneme vyšší výpočetní sílu. Ta spočívá v tom, že budeme moci kontrolovat i gramatiky, které již nejsou bezkontextovými gramatikami, ale ještě nemůžeme kontrolovat všechny kontextové gramatiky.

3 Hluboké zásobníkové automaty

3.1 Neformální definice

Hluboký zásobníkový automat umožňuje rozvíjet, oproti klasickému zásobníkovému automatu, neterminální symboly i v jiné hloubce než je hloubka 1, která značí vrchol zásobníku. Z toho vyplývá, že můžeme zajistit pomocí vhodných pravidel, aby se postupně rozvinuly i neterminální symboly v zásobníku, které nejsou přímo na jeho vrcholu.

Pravidla pro hluboký zásobníkový automat se proto skládají ještě z čísla, které určuje, v jaké hloubce zásobníku dojde k rozvoji symbolu za řetězec, který je určen v pravidle. Tato hloubka musí být menší nebo rovna maximální hloubce automatu, ve které je automat schopen rozvíjet symboly.

3.2 Formální definice

Hluboký zásobníkový automat je definován sedmicí symbolů:

$$_{deep}M = (Q, \Sigma, \Gamma, R, s, S, F)$$

deep maximální hloubka automatu

Q množina přípustných stavů, ve kterých se automat může nacházet

$\sum$ vstupní abeceda symbolů

Γ abeceda zásobníkových symbolů, kde $\Gamma \supseteq \Sigma, \# \in (\Gamma - \Sigma)$ a # je speciální symbol označující dno zásobníku

R množina pravidel tvaru:

$mpA \longrightarrow qv$

- *m* - hloubka, ve které se pravidlo použije
- *p* - stav, ve kterém se automat před rozvojem nachází
- *A* - neterminální symbol, který se bude nahrazovat
- *q* - stav, do kterého automat po použití pravidla přejde
- *v* - řetězec, kterým bude neterminální symbol nahrazen $v \in \Gamma^*$

s startující stav, stav, ve kterém se bude automat nacházet na začátku rozvoje

$s \in Q$

S startující symbol v zásobníku. Neterminální symbol, který se bude na začátku rozvoje nacházet na vrcholu zásobníku

$S \in \Gamma$

F množina koncových stavů, stavů, kdy se řetězec bere již za rozpoznaný, akceptovaný

$F \subset Q$

3.3 Výpočetní krok

Jako výpočetní krok je brán přechod z jednoho stavu do druhého za použití jednoho z pravidel stanovených v konfiguraci automatu. Výběr pravidla, pomocí kterého přejde automat z jednoho stavu do druhého, je proveden v závislosti na aktuálním stavu automatu, aktuální maximální hloubce a neterminálních symbolech v jednotlivých hloubkách.

Každý výpočetní krok můžeme zapsat pomocí dvojice stavů. Každý stav se dá prezentovat jako trojice regulárních výrazů.

Stav automatu:

$$(Q; \Sigma^*; (\Gamma - \{\#\})^*\{\#\})$$

Q - stav, ve kterém se automat nachází, nebo bude nacházet

$\sum^*$ - vstupní řetězec

$(\Gamma - \{\#\})^*\{\#\}$ - obsah zásobníku, který je ukončen symbolem # označujícím dno zásobníku

Příklad:

$$(s; aabbcc; S\#)$$

Pokud chceme znázornit krok výpočtu, zapíšeme tento přechod pomocí relace dvojice stavů s naznačením, že automat přechází z prvního do druhého. Toto naznačíme pomocí šipky orientované od výchozího stavu do následného.

$$(s; aabbcc; S\#) \Rightarrow (p; aabbcc; AA\#)$$

Pokud bychom chtěli zjednodušit výpočetní krok z několika kroků na jeden, můžeme k tomu využít tranzitivní uzávěr relace přechodu, který nám naznačuje, že se z výchozího stavu můžeme dostat do následujícího pomocí nedefinovaného počtu přechodů, které však vychází z pravidel daných v konfiguraci automatu.

$$(s; aabbcc; S\#) \Rightarrow^* (f, \varepsilon, \#)$$

Pokud najdeme posloupnost kroků, které nás dovedou do stavu, který patří do množiny stavů koncových, přečteme celý vstupní řetězec a podaří se nám vyprázdnit celý zásobník, znamená to, že tato věta patří do jazyka generovaného gramatikou, která je ekvivalentní našemu automatu. Pokud si uložíme posloupnost použitých pravidel, které jsme použili k přechodu z výchozího stavu do koncového, můžeme sestavit zpětně syntaktický strom, pomocí kterého můžeme poté zpětně složit danou větu, u které jsme se rozhodovali, zda přijmeme či nikoli.

Někdy vyžadujeme naopak vytvořit syntaktický strom tzv. metodou zdola nahoru, což zajistíme obrácením pořadí použitých pravidel a jejich otočením. Poté dostaneme pravidla, která nepřevádějí nonterminály na terminály, ale právě terminály na nonterminály. To se nám může hodit

zejména tehdy, chceme-li provést syntaktický strom pomocí postupné generalizace jednotlivých úkonů. To se může použít například u překladačů, nebo interpretů k zajištění efektivnějšího a úspornějšího kódu.

3.4 Příklad rozvoje

Mějme hluboký zásobníkový automat hloubky 2 popsaný následovně

$$_2M = (Q, \Sigma, \Gamma, R, s, S, \{f\})$$

$$\begin{aligned}
Q &= \{s, p, q, f\} \\
\Sigma &= \{a, b, c\} \\
\Gamma &= \{a, b, c, A, S, \#\} \\
R &= \{\ 1sS \rightarrow qAA, \\
&\quad\ 1qA \rightarrow paAb, \\
&\quad\ 1qA \rightarrow fab, \\
&\quad\ 2pA \rightarrow qAc, \\
&\quad\ 1fA \rightarrow fc\ \}
\end{aligned}$$

Výše definovaný automat přijímá věty z gramatiky $a^n b^n c^n$. V následujícím příkladu ilustruji, jak probíhá přijímání vstupního řetězce *aabbcc*.

$$\begin{aligned}
(s, aabbcc, S\#) &\Longrightarrow (q, aabbcc, AA\#) && [1] \\
&\Longrightarrow (p, aabbcc, aAbA\#) && [2] \\
&\Longrightarrow (p, abbcc, AbA\#) \\
&\Longrightarrow (q, abbcc, AbAc\#) && [4] \\
&\Longrightarrow (f, abbcc, abbAc\#) && [3] \\
&\Longrightarrow (f, bbcc, bbAc\#) \\
&\Longrightarrow (f, bcc, bAc\#) \\
&\Longrightarrow (f, cc, Ac\#) && [5] \\
&\Longrightarrow (f, cc, cc\#) \\
&\Longrightarrow (f, c, c\#) \\
&\Longrightarrow (f, \varepsilon, \#)
\end{aligned}$$

Na tomto příkladu je zřejmé, jak tento automat bude fungovat. Nejzajímavější je pravidlo číslo 4, které je právě podstatou těchto hlubokých zásobníkových automatů. Rozvíjí 2. neterminální symbol, což je v pravidle naznačeno číslem 2, před samotným přepisovacím pravidlem. Pokud bychom neuměli pracovat s neterminálními symboly uvnitř zásobníku, nikdy bychom tohoto nemohli docílit a tudíž, ani vytvořit takový automat, který by mohl s jistotou označit řetězec za patřící do jazyka generovaného stejnou gramatikou, jakou tento automat přijímá.

4 Implementace

Jako implementační prostředí syntaktického analyzátoru s hlubokými zásobníky jsem použil vývojové prostředí C# s platformou .NET™. Bylo to z důvodu jednouché implementace funkčnosti a již předdefinovaných tříd, které se daly jednoduše používat. Jedná se zejména o generické seznamy, které nabízejí velkou funkčnost při práci s dynamickými strukturami o různém základu. Tyto generické seznamy implementují v sobě již odladěné funkce, které většinou pracují efektivněji, než způsoby při použití vlastních algoritmů.

Téměř celý program je koncipován pomocí OOP. Je zde použito několik navzájem navazujících tříd. Většinou se jedná o třídy, které byly vytvořeny z důvodu zapouzdřenosti některých hodnot, u kterých je zadávání hodnot ošetřeno právě pomocí metod, které slouží jako rozhraní pro ostatní funkce, které tyto objekty budou používat.

Všechny třídy jsem pojmenoval s předponou „C_“ od anglického Class, aby bylo na první pohled jasné, že se jedná o třídu, nikoli o identifikátor, či funkci. Většinu vlastností jsem pojmenovával anglickými názvy, aby nedocházelo k problémům v rozlišení českých názvů bez diakritiky.

4.1 C_XMLwork

Tato třída slouží k načtení XML dokumentu. Tato třída obsahuje pouze jednu funkci a to ReadXML(), která načte soubor, jehož cesta se funkci předá v parametru. Funkce se pokusí otevřít dokument a přečíst z něj kořenový prvek – root element, který je uzavřen do páru značek <deeppushdownautomaton> a </deeppushdownautomaton>. Pokud se nepodaří soubor najít, nebo se nepodaří najít kořenový prvek, odchytí vzniklou chybu, kterou generuje třída XmlTextReader při pokusu otevřít stream, který bude obsahovat obsah souboru připravený k parserování. Chybu může také vygenerovat parserovací třída XmlDocument. Všechny tyto chyby jsou odchyceny pomocí bloku try, catch a pokud některá chyba nastane, funkce vrátí hodnotu *null*, což značí chybu při načítání. Proto bychom měli při volání této funkce testovat zpětně, co nám vrací.

4.2 C_Symbol

Třída C_Symbol je zapouzdřením struktury, která obsahuje definici symbolů, které mohou být použity ať už v zásobníku, vstupním řetězci, nebo i v pravidle. Struktura má dvě privátní položky, které obsahují název symbolu, jeho definici např. pomocí řetězce, a dále znakovou proměnnou, která v sobě uchovává typ tohoto znaku. Tento typ značí, či se tento znak jeví jako terminální, či neterminální. Pokud je znak terminální, je zde uložen znak ‘T’ v opačném případě je zde znak ‘N’.

K těmto položkám se přistupuje přes veřejné položky, které zveřejňují jejich hodnoty pouze pro čtení. Naplnění těchto hodnot se provádí při inicializaci instance této třídy. Konstruktor vyžaduje dva parametry, přičemž první je hodnota prvku a druhý parametr je typ symbolu.

Tato třída obsahuje ještě dvě metody. První metoda plně překrývá původní metodu odvozenou od třídy Object, která je rodičovskou třídou pro všechny třídy, a to metodu ToString(). V této třídě tato metoda vrací řetězec, který je ve tvaru *název.typ*.

Druhá metoda je přetížením metody Equals(). V tomto případě metoda porovnává svou instanci s jinou instancí předanou v parametru. Tato instance musí být stejné třídy, nikoli jako v původní verzi libovolným objektem. Metoda vrací hodnotu *true*, jestliže se tyto objekty shodují v názvu i typu, jinak vrací hodnotu false.

4.3 C_Alphabet

Třída C_Alphabet zapouzdřuje generický seznam tříd C_Symbol. Tato třída rozšiřuje tento seznam o metodu ContainSymbol(), která vrací hodnotu *true*, jestliže při průchodu narazí na symbol daný parametrem. Pokud projde celý seznam a na daný prvek nenarazí, vrací hodnotu false. Dále seznam rozšiřuje o funkce Print a Print2, které vypisují na streamovaný výstup, například do souboru, všechny znaky obsažené v této abecedě. Metoda ToString je zde opět překrytá vlastní funkcí. Tato funkce vrací do řetězce výpis všech prvků v abecedě. K tomu využívá u každého prvku jeho překrývající funkci ToString.

4.4 C_Subrule

Tato třída byla přidána, aby mohla být implementována paralelní verze hlubokého zásobníkového automatu Petera Solára. Tato třída obsahuje pravou, a levou stranu vždy jednoho atomického pravidla, ze kterých se v paralelní verzi pak skládají všechna pravidla, která mohou být v automatu použita.

Obsahuje tři privátní položky a to symbol, který se bude expandovat, řetězec, na který bude expandováno, a hloubku, ve které k expanzi dochází. Zveřejněné rozhraní umožňuje pouze tyto hodnoty číst, nikoli měnit. Inicializace privátních položek se provádí při vytváření instance třídy v konstruktoru. Konstruktoru se předávají parametry všech tří položek a těmito hodnotami se při vytváření naplní. Za běhu programu se již nedají měnit.

4.5 C_Rule

Tato třída slouží k zapouzdření seznamu instancí C_Subrule. Jedno pravidlo se pak skládá z několika subpravidel. Toto je implementováno z důvodu kompatibility s paralelní verzí. Pokud se jedná o pouze neparalelní verzi, obsahuje seznam pouze jedno subpravidlo.

Všechny prvky jsou implementovány jako privátní s veřejným rozhraním, které umožňuje vesměs pouze čtení jejich hodnot. Toto zajišťuje stabilitu při provádění expanzí. Tj. nemůže se stát, že se pravidlo změní během běhu automatu. Načítání pravidel se provádí inicializací v konstruktoru.

Třída dále obsahuje číslo pravidla, což je pořadím pravidla při načítání z XML. Je pak identifikovatelné podle tohoto čísla a slouží k zápisu použitých pravidel.

Dalším prvkem jsou stavy, ve kterých se musí automat nacházet před rozvojem, a do kterého přejde automat po dokončení rozvoje. Tyto stavy se inicializují již v konstruktoru a nelze je později změnit. Získat je pak můžeme pomocí veřejných rozhraní *State1* a *State2*, která jsou jen pro čtení.

Do pravidla můžeme přidat subpravidlo pomocí metody Add. Ta přidá do seznamu subpravidel to, které bylo předáno v parametru metody. Toto slouží zejména při načítání pravidel ze souboru XML. Pokud se jedná o neparalelní verzi, tak pozdější přidávání subpravidel stejně neovlivní výsledek, jelikož nám vrací stejně vždy pouze to, které má v seznamu index [0].

Je zde rozhraní CountRules, které vrací počet subpravidel, pokud má pravidlo příznak paralelní, nebo vrací hodnotu 1 pro neparalelní verzi. Dále je zde funkce ToString, která, jako v každé třídě, vypisuje v našem formátu obsah instance třídy. Poslední funkcí, která je pro tuto třídu zajímavá je funkce GetSubrule. Ta nám vrací pravidlo, které leží na indexu daném jí parametrem. Provede zde také kontrolu, zda například nepožadujeme subpravidlo *n*, pokud nemáme paralelní verzi automatu. V takovém případě vrátí funkce hodnotu *null*. Je to efektivnější, než kdyby měla generovat chybu. Hodnotu *null* vrací také v případě, že požadujeme subpravidlo v pořadí, které v seznamu není.

4.6 C_Rules

Tato třída zapouzdřuje seznam pravidel. Vytváří jim rozhraní a přidává funkce specifické pro práci s vyhledáváním v pravidlech. Obsahuje sice pouze jeden prvek, který je veřejně přístupný, ale zapouzdřuje funkce potřebné pro práci s pravidly jako takovými.

Většina funkcí vrací objekt typu C_Rules. Každá funkce vytvoří novou instanci třídy C_Rules, tu naplní pravidly ze svého seznamu, které splňují podmínky dané pro danou funkci. Funkce FindRulesByState vrací seznam pravidel, která vycházejí z daného stavu. Tato funkce projde celý svůj vnitřní seznam, a pokud u některého z pravidel se vstupní stav shoduje se stavem předaným v parametru, přidá toto pravidlo do dočasného seznamu, který vytvořil na začátku. Nakonec vrátí tento seznam.

Funkce GetRulesBy vrací opět nově vytvořený seznam, tentokrát si najde pravidlo, které je určeno číslem v parametru funkce a vrátí všechna pravidla, která za tímto pravidlem následují. Toto je použito, když provádíme Rollback pravidel, z důvodu, aby nepoužil stejné pravidlo, ale pravidlo následující.

4.7 C_State

Třída zaobaluje stavy, do kterých se automat může dostat. Každý stav má svůj název a logické proměnné, zda je tento stav výchozí a jestli je koncový. Pokud je stav koncový, znamená to, že se automat dostal do stavu, kdy může prohlásit, že řetězec, který měl na vstupu, náleží do jazyka generovaného gramatikou, ale pouze tehdy, je-li zásobník vyprázdněn a je přečten celý vstupní řetězec.

Dále je zde přetížená metoda Equals, která porovnává stavy podle shodnosti všech položek a vrací hodnotu *true*, pokud jsou všechny položky obou objektů totožné hodnotou, nikoli adresou. Nakonec je zde ještě překrytá metoda ToString.

4.8 C_States

Třída obsahuje jeden prvek generický seznam typu C_State. V tomto seznamu jsou uloženy všechny stavy, které jsou povoleny z konfigurace. Využívá se toho při načítání konfigurace z XML souboru, aby se mohlo určit, zda je konfigurace správná, či nikoli.

Funkce GetStart vrací startovací stav, ze kterého se vychází. Tato funkce projde postupně celý seznam stavů, a jakmile najde stav, který má proměnnou start v hodnotě *true*, ukončí provádění a vrátí ukazatel na tento stav. Funkce FindState vrací ukazatel na stav podle jeho názvu.

4.9 C_PushDown

Nejdůležitější třída celého programu zapouzdřuje zásobník. Tato třída má jeden soukromý prvek, který je implementován jako generický seznam. Důležitější jsou funkce, které s touto strukturou pracují.

První funkcí, která stojí za zmínku je funkce GetFirst, která vrátí ukazatel na první prvek vnitřního seznamu, pokud však je seznam prázdný, vrátí hodnotu *null*.

Další funkcí je funkce Pop, která vyjme první prvek ze seznamu a vrátí na něj ukazatel, jinak vrátí hodnotu *null*. Vrácení ukazatele je pouze formální, aby se eventuelně dalo kontrolovat, zda není zásobník prázdný.

Funkce MaxDeep vrací počet neterminálních symbolů v zásobníku.

Funkce InDeep vrací ukazatel na neterminální symbol v hloubce předané jí parametrem. Prochází celým zásobníkem od shora dolů, a když narazí na neterminální symbol, zvýší svůj čítač o 1. Pokud se čítač shoduje s parametrem funkce, vrátí ukazatel tohoto prvku. Pokud projde celý zásobník, ale nenarazí na dostatečný počet neterminálních symbolů, vrátí *null*.

Nejdůležitější funkcí je funkce Expand. Tato funkce si zjistí ukazatel na prvek v hloubce, která je jí předána. Pokud je ukazatel jiný než *null*, což značí úspěch, najde si v seznamu index tohoto prvku. Poté odebere tento symbol a na jeho místo vloží posloupnost symbolů, které jsou předány pomocí parametru *By*. Tento parametr je implementován jako seznam symbolů. Jakmile odebere neterminální symbol, vloží na jeho místo první prvek z *By*. Za něj pak vkládá postupně ostatní prvky z tohoto seznamu.

Dále jsou zde implementovány funkce, které jsou použity k operacím backtrackingu. Jsou to funkce Export a load. Funkce Export vytvoří nový seznam, do kterého vloží aktuální pořadí prvků, které jsou v zásobníku obsaženy. Tento seznam poté použije jako návratovou hodnotu.

Funkce load načte zásobník do stavu, který je předán funkci jako parametr. Tento stav je implementován, jako seznam symbolů, které se nacházely v zásobníku při operaci Export.

Poslední funkcí je funkce GetString, která vypíše obsah zásobníku do řetězce. Vypisuje jej ale v opačném pořadí. Tudíž vrchol zásobníku je na pravé straně řetězce. Toto se hodí právě do našeho programu, kdy se nám vypisuje zásobník na levé straně automatu a tudíž je pak názornější, když se symbol na vstupu shoduje se symbolem na vrcholu zásobníku.

4.10 C_PDAConf

Tato třída slouží k načtení a pozdějším zjišťováním výchozí konfigurace automatu. Tato třída představuje pomyslnou sedmici definující hluboký zásobníkový automat. Je zde hlavní metoda XMLinput, která zpracovává XML soubor načtený pomocí třídy C_XMLwork. Všechny položky jsou privátní s veřejným rozhraním, které umožňuje tyto hodnoty pouze číst, nikoli měnit.

Funkce XMLinput nejprve vytvoří pomocí třídy C_XMLwork, dokument XML, který následně začne parserovat pomocí vestavěných funkcí jazyka C# a rozhraní .NET™. Jako první použije funkci SelectSingleNode, která vrátí pouze jeden jediný kořen zadaný parametrem. Tímto kořenem je nyní *deeppushdownautomaton/states*. Z tohoto kořene načte všechny stavy a načte je do své privátní proměnné. Takto načte vstupní abecedu, zásobníkovou abecedu, pravidla a nakonec i vstupní řetězec, který chceme zjistit, zda náleží do jazyka generovaného gramatikou, kterou načetl v podobě pravidel výše.

K ostatním položkám třídy je možno přistupovat pomocí veřejného rozhraní. Nejčastěji budeme přistupovat k položce Rules.

4.11 C_Config

V této třídě probíhá téměř veškerá funkčnost celého automatu. Je zde uložena aktuální konfigurace automatu, jak ji známe ze zápisu v kapitole 3.3. Je zde uložen aktuální obsah zásobníku, stav, ve kterém se automat nachází, a řetězec na vstupu, který je třeba zkontrolovat.

Obsah zásobníku je uložen v instanci třídy C_PushDown, která nám tím nabízí metody jako je Expand, Export a load. Stav automatu je uložen v instanci třídy C_State. Vstupní řetězec je uložen v instanci třídy C_Alphabet.

V konstruktoru je třeba uvést základní nastavení automatu, tzn. Nastavit, v jakém stavu se nachází, jaké symboly má v zásobníku a jaký řetězec se nachází na vstupu. Toto všechno lze získat z instance třídy C_PDAConf. Nejpoužívanější konstruktor by mohl vypadat například takto:

```
C_Config Aut = new C_Config(new C_PushDown(PDACon.GammaNoSigma.retezec[0]),
                PDACon.Input.retezec, PDACon.States.GetStart());
```

V tomto příkladu jsme vytvořili konfiguraci automatu, lépe řečeno stav, ve kterém se automat nachází. Použili jsme k tomu konstruktor C_Config a dále jsme předpokládali, že jsme konfiguraci načetli do Objektu PDACon, ze kterého jsme poté získávali další konfigurační informace potřebné k vytvoření prvního stavu automatu.

Další důležitou funkcí je UseRule, která nám rozhodne, zda pravidlo, které jí předáme jako parametr, může být použito, tj. splňuje podmínky, že automat musí být ve stavu, který odpovídá stavu na levé straně pravidla, dále zda v hloubce, která je definovaná v pravidle, je neterminální symbol, který je v pravidle na levé straně. Pokud všechny tyto podmínky jsou splněny, dojde k použití funkce Expand z třídy zásobníku a předají se jí parametry, vyjmuté z pravidla, které bylo této funkci předáno. Pokud se rozvoj pravidla podaří, přejde automat do dalšího stavu a funkce vrátí hodnotu *true*. Pokud se rozvoj nepodaří, nebo není splněno některé z předcházejících pravidel, vrátí funkce hodnotu false.

Další důležitou funkcí je funkce UseableRules. Této funkci se předá seznam pravidel, která splňují podmínku, že musí být automat v určitém stavu před rozvojem. Tato funkce poté projde všechna pravidla a vytvoří seznam nový, do kterého vloží pouze ta pravidla, která mohou být použita v závislosti na ostatních podmínkách pro rozvoj. Zkontroluje, zda se v hloubkách daných pravidly nachází ty neterminální symboly, které jsou na levé straně pravidel.

K zajištění funkčnosti backtrackingu, který je používán při rozhodování nedeterministických pravidel, se používají funkce BackUp a GetBack. Tyto funkce pracují s proměnnými, které jsou privátní pro tuto třídu. V těchto proměnných se ukládá do seznamů obsah zásobníku, vstupní řetězec, stav, ve kterém se automat nachází a pravidlo, které bylo použito v nedeterministickém stavu. Funkce BackUp přidá do seznamů další položky představující stav proměnných v době zavolání funkce. Oproti tomu funkce GetBack načte z těchto seznamů vždy poslední vložený prvek, načte jej do příslušné proměnné a tento prvek ze seznamu odebere.

Poslední funkce, o kterých se tu zmíním, jsou GetZasobnikToStr a GetVstupToStr. Tyto funkce se používají ke znázornění obsahu zásobníku a vstupního řetězce. Jakmile se obsah zásobníku znázorní, je lépe pochopitelné, jak tento automat funguje. Bude i lépe názorné, že dochází k rozvoji hlouběji v zásobníku, než jen na jeho vrcholu.

4.12 Interaktivní funkčnost

Všechny tyto funkce jsou implementovány pod tlačítko Další. Ve třídě Program jsou nadefinovány globální proměnné Automat a Konfigurace. Díky tomu můžeme k těmto proměnným přistupovat z kterékoli třídy.

Jakmile uživatel v prvním okně aplikace vybere soubor s konfigurací a načte jej, je automaticky přesměrován na druhý formulář. V něm se načte konfigurace do předpřipravených polí.

Jakmile uživatel stiskne tlačítko Další, zavolá se funkce b_next_click. V této funkci nejprve dojde ke zjištění, zda náhodou není zásobník vyprázdněn. Pokud je, zkontroluje se, zda není prázdný i vstupní řetězec a zda se automat nenachází v některém z koncových stavů. Pokud by zásobník byl prázdný, byl prázdný i vstup a automat se nacházel v koncovém stavu, tak řetězec, který byl na vstupu, patří do jazyka generovaného gramatikou *G*. Pokud se automat nedostal při vyprázdnění do koncového stavu, nebo nebyl přečten celý vstupní řetězec, provede se rollback. Rollback spočívá v zavolání funkce GetBack a nastavení příznaku *err*. Pokud tato funkce vrátí hodnotu false, značí to, že se nelze vrátit do žádného předchozího stavu. Tudíž řetězec nemůže být přijat automatem.

Pokud zásobník není prázdný, pokračuje funkce testováním, zda není symbol na vrcholu zásobníku terminálním symbolem. Pokud ano, porovná tento symbol se symbolem na vstupu. Jestliže jsou tyto symboly shodné, provede Pop zásobníku a načtení dalšího vstupního symbolu. Jestliže nejsou tyto symboly shodné, provede rollback.

V případě, že na vrcholu zásobníku je neterminální symbol, začne se hledat vhodné pravidlo pro rozvoj neterminálních symbolů v zásobníku. Výběr pravidla je založen zejména na výběru podle stavu, ve kterém se automat právě nachází. Toto provede funkce UseableRules. Ta vrátí seznam pravidel, která se mohou použít k expanzi neterminálních symbolů. Jestliže je nastaven příznak *err*, je proveden ještě výběr pravidel pomocí funkce GetBulesBy, která vrátí pouze pravidla, která ještě nebyla použita v tomto stavu. Tato pravidla jsou vypsána do políčka, které se nachází vpravo nahoře. Tento výpis se provádí z důvodu kontroly, aby si uživatel mohl prohlédnout, která pravidla mohou být v tomto stavu použita. Pokud funkce vrátí seznam, který neobsahuje žádné prvky, znamená to, že jsme se dostali do stavu, kdy nemůžeme pokračovat v rozvoji, tudíž provede rollback za současného vypsání „Backtracking“ na místo nabízených pravidel. Pokud existuje pouze jedno pravidlo, je použito pomocí funkce UseRule a číslo pravidla, které bylo použito, se přidá na konec seznamu použitých pravidel, který se nachází vlevo dole.

Jestliže se však stane, že existuje více než 1 pravidlo, které je možno použít, provede se BackUp stavu automatu a pak expanze pomocí prvního pravidla ze seznamu.

Pokud nás nezajímá celý proces testování stavů, můžeme použít tlačítko Automatic, které provádí v cyklu volání předchozí funkce do té doby, než dojde do stavu, kdy zjistí, že buď byl řetězec přijat, nebo nelze daný řetězec přijmout.

Pokud byl řetězec rozpoznán, objeví se v políčku pro výpis nabízených pravidel text „Heuráka“ a v seznamu použitých pravidel si můžeme přečíst posloupnost pravidel, která byla použita k tomu, aby byl řetězec akceptován. Pokud tato pravidla zpětně použijeme, měli bychom dostat zpět řetězec, který jsme se snažili rozpoznat.

5 Ilustrativní příklady

5.1 Klasický zásobníkový automat

Mějme klasický zásobníkový automat

$$M = (Q, \Sigma, \Gamma, R, s, S, \{f\})$$

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

Σ = $\{a, b, c, d\}$

Γ = $\{a, b, c, d, A, B, S, \#\}$

R = $\{ sS \rightarrow pAB,$

$pA \rightarrow paAb,$

$pA \rightarrow qab,$

$qB \rightarrow qcBd,$

$qB \rightarrow fcd \}$

Tento automat bude kontrolovat gramatiku $a^n b^n c^m d^m$, u těchto automatů můžeme kontrolovat pouze 2 po sobě jdoucí terminální symboly, zda se opakují ve skupinách o stejném počtu, pokud bychom chtěli kontrolovat například 3 skupiny symbolů, zda mají v každé skupině stejný počet znaků, tento aparát nám k tomu nestačí.

$(s, aaabbbccdd, S\#) \Longrightarrow (p, aaabbbccdd, AB\#)$ [1]

$\Longrightarrow (p, aaabbbccdd, aAbB\#)$ [2]

$\Longrightarrow (p, aabbbccdd, AbB\#)$

$\Longrightarrow (p, aabbbccdd, aAbbB\#)$ [2]

$\Longrightarrow (p, abbbccdd, AbbB\#)$

$\Longrightarrow (q, abbbccdd, abbbB\#)$ [3]

$\Longrightarrow (q, bbbccdd, bbbB\#)$

$\Longrightarrow (q, bbccdd, bbB\#)$

$\Longrightarrow (q, bccdd, bB\#)$

$\Longrightarrow (q, ccdd, B\#)$

$\Longrightarrow (q, ccdd, cBd\#)$ [4]

$\Longrightarrow (q, cdd, Bd\#)$

$\Longrightarrow (f, cdd, cdd\#)$ [5]

$\Longrightarrow (f, dd, dd\#)$

$\Longrightarrow (f, d, d\#)$

$\Longrightarrow (f, \varepsilon, \#)$

Z tohoto příkladu je patrno, že nemůžeme zajistit, aby se testovala gramatika například $a^n b^n c^n d^n$, protože nemáme mechanismus, jak zajistit, aby se vždy rozvinuly oba neterminální symboly v zásobníku.

5.2 Hluboký zásobníkový automat

Mějme gramatiku $a^n b^{2n} c^n$ a automat, který tuto gramatiku umí rozpoznat. Na vstup mu dáme řetězec *aaabbbbbbccc* a znázorníme, jak bude automat postupovat při jeho rozpoznávání.

$$_2M = (Q, \Sigma, \Gamma, R, s, S, \{f\})$$

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

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

$\Gamma = \{a, b, c, A, S, \#\}$

$R = \{\ 1sS \to qAA,$

$1qA \to paAbb,$

$1qA \to fabb,$

$2pA \to qAc,$

$1fA \to fc\ \}$

$(s, aaabbbbbbccc, S\#) \Longrightarrow (q, aaabbbbbbccc, AA\#)$ [1]

$\Longrightarrow (p, aaabbbbbbccc, aAbbA\#)$ [2]

$\Longrightarrow (p, aabbbbbbccc, AbbA\#)$

$\Longrightarrow (q, aabbbbbbccc, AbbAc\#)$ [4]

$\Longrightarrow (p, aabbbbbbccc, aAbbbbAc\#)$ [2]

$\Longrightarrow (p, abbbbbbccc, AbbbbAc\#)$

$\Longrightarrow (q, abbbbbbccc, AbbbbAcc\#)$ [4]

$\Longrightarrow (f, abbbbbbccc, abbbbbbAcc\#)$ [3]

$\Longrightarrow (f, bbbbbbccc, bbbbbbAcc\#)$

$\Longrightarrow (f, bbbbbccc, bbbbbAcc\#)$

$\Longrightarrow (f, bbbbccc, bbbbAcc\#)$

$\Longrightarrow (f, bbbccc, bbbAcc\#)$

$\Longrightarrow (f, bbccc, bbAcc\#)$

$\Longrightarrow (f, bccc, bAcc\#)$

$\Longrightarrow (f, ccc, Acc\#)$

$\Longrightarrow (f, ccc, ccc\#)$ [5]

$\Longrightarrow (f, cc, cc\#)$

$\Longrightarrow (f, c, c\#)$

$\Longrightarrow (f, \varepsilon, \#)$

Jak je vidět z příkladu, mají tyto automaty větší sílu. Díky těmto automatům můžeme identifikovat některé z řetězců náležícím ke kontextovým gramatikám. Ne však všechny, tudíž tyto automaty dokáží identifikovat gramatiky, které leží za hranicí bezkontextových gramatik.

6 Závěr

V této práci jsem se zaměřil na implementaci hlubokého zásobníkového automatu. Tento automat má větší sílu, tudíž může obsáhnout více gramatik. Gramatiky určíme správně zavedenými pravidly. Výsledkem této práce je syntaktický analyzátor, založený na hlubokých zásobníkových automatech. Tyto automaty se mohou využít v překladačích, při syntaktické analýze, nebo při syntaxí řízeném překladu. Tento syntaktický analyzátor může však být použit i v jiných odvětvích teoretické informatiky. Lze jej použít všude tam, kde je třeba určit, zda řetězec, který je na vstupu, je správně zapsán, či nikoli. Tudíž můžeme rozlišit správnost zapsaného řetězce.

Dalším rozšířením tohoto automatu je paralelní verze tohoto automatu, kterou se zabýval kolega Peter Solár ve své práci. Tyto automaty mají stejnou generativní sílu, jen paralelní verze je schopna provést více rozvojů během jedné časové jednotky, tudíž pracuje rychleji.

Tuto práci můžeme dále rozšiřovat. Příklad použití může být syntaxí řízený překlad, jímž se budu zabývat v navazující práci.

Literatura

[1] Meduna, A. Deep pushdown automata. Acta Informatica, 98, 2006,

[2] Češka, M. Teoretická infromatika, učební texty, FIT VUT v Brně 2002

[3] Meduna, A. Automata and Languages,London, Springer, 2000

Seznam příloh