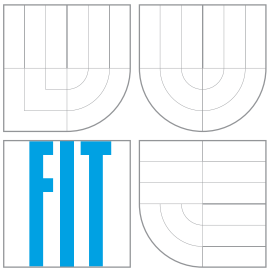
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA AUTOMATECH S HLUBOKÝMI ZÁSOBNÍKY

PARSING BASED ON AUTOMATA WITH DEEP PUSHDOWNS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DAVID RUSEK

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. RNDr. ALEXANDER MEDUNA, CSc.

BRNO 2016

Abstrakt

Tato práce se zabývá problematikou návrhu a implementace syntaktické analýzy založené na kontextových jazycích, obsahujících konstrukce, které není možné analyzovat pomocí standardních syntaktických analyzátorů založených na bezkontextových gramatikách. Konkrétně se tato práce zabývá možností rozšíření LL-analýzy o kontextovou podporu a to nahrazením klasických zásobníkových automatů pomocí hlubokých zásobníkových automatů (HZA), tak jak je představil a popsal prof. Alexander Meduna.

Abstract

This paper addresses the issue of design and implementation of syntactic analysis based on the context sensitive languages, respectively, grammars that contains constructs, which isn't possible to analyze with the help of the standard parsers based on the context free grammars. More specifically, this paper deals with the possibility of adding context sensitive support to the classic LL-analysis by replacing the standard pushdown automata (PDA) with deep pushdown automata (DP), which were introduced and published by prof. Alexander Meduna.

Klíčová slova

Syntaktická analýza, Zásobníkové automaty, Kontextové jazyky, Hluboké zásobníkové automaty.

Keywords

Syntactic Analysis, Parsing, Parsers, Pushdown Automata, Deep Pushdown Automata, PDA, PD, Context Sensitive Languages, CSL, CSG.

Citace

David Rusek: Syntaktická analýza založená na automatech s hlubokými zásobníky, diplomová práce, Brno, FIT VUT v Brně, 2016

Syntaktická analýza založená na automatech s hlubokými zásobníky

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Alexandra Meduny.

.....
David Rusek
24. května 2016

Poděkování

Zde bych rád poděkoval prof. Alexandru Medunovi za jeho odbornou pomoc při řešení diplomové práce.

© David Rusek, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Drtivá většina všech dnešních programovacích jazyků je kontextových, paradoxně jejich syntaktické analyzátory jsou založené na bezkontextových gramatikách a zásobníkových automatech, které dokáží zpracovávat pouze bezkontextové jazyky. Proto jsou překladače obecně doplněny řadou rozšíření, ze kterých je zřejmě nejdůležitější tabulka symbolů, která v konjunkci s bezkontextovým syntaktickým analyzátorem je schopna obsáhnout kontextuálnost těchto jazyků. Nicméně i tento přístup má svá omezení, jelikož tabulka symbolů slouží především k analýze sémantického charakteru a neumožňuje provádět kontextovou analýzu syntaktického charakteru.

Tato práce se zabývá návrhem syntaktické analýzy založené na hlubokých zásobníkových automatech, jenž představil prof. Alexander Meduna [6], a které pracují na stejném principu jako klasické zásobníkové automaty s tím rozdílem, že mohou expandovat i neterminální symboly hlouběji na zásobníku. Jejich síla, neuvažujeme-li ε-pravidla, odpovídá síle stavových gramatik, které definoval Takumi Kasai v roce 1970 [9], což znamená, že jsou stejně jako stavové gramatiky schopny alespoň teoreticky obsáhnout nekonečnou hierarchii jazyků mezi třídami jazyků bezkontextových a kontextových.

Vzpomeňme si na klasický zásobníkový automat, který při každém kroku může provést buď operaci *compare* (při výskytu terminálního symbolu na vrcholu zásobníku) nebo operaci *expand* (při výskytu neterminálního symbolu na vrcholu zásobníků). Hluboký zásobníkový automat funguje totožně, pouze s tím rozdílem, že pokud je na vrcholu zásobníku neterminál, hluboký ZA umožňuje expandovat i neterminály mimo vrchol zásobníků. Tedy, zatímco operace *compare* zůstane nezměněna, operace *expand* bude rozdílná.

Zcela jistě existuje mnoho přístupů jak využít hluboké zásobníkové automaty pro kontextovou analýzu, nicméně tato práce se konkrétně bude zabývat možností rozšíření klasické LL-analýzy pracující ve směru shora-dolů, náhradou standardního zásobníkového automatu za hluboký, přičemž se primárně zaměřuje na možnost obohacení klasických programovacích jazyků o pokročilé, kontextové, syntaktické konstrukce.

V druhé kapitole tato práce formálně definuje stěžejní pojmy jako jsou: gramatika, LL-gramatika, stavová gramatika, zásobníkový automat a hluboký zásobníkový automat a některé další související pojmy. Zároveň bude uveden jednoduchý příklad práce hlubokého ZA.

Další kapitola pak detailně zkoumá problémy související s návrhem gramatik (stavových i těch klasických), které plynou z vyřazení ε-pravidel.

Čtvrtá kapitola pokračuje návrhem změn standardní LL-analýzy potřebných pro zahrnutí hlubokého ZA, přičemž zde bude popsána především rozdílná struktura LL-tabulek pro klasickou a hlubokou formu LL-analýzy, algoritmy pro výpočet množin hloubek pravidel sta-

vové gramatiky a algoritmy pro výpočet množin *First* a *Follow* pro hlubokou LL-analýzu. Nezapomeneme však ani na důležitou kapitolu věnovanou zotavení z chyb.

V páté kapitole budou shrnuty výsledky kapitol dva až čtyři a to formou formální definice pojmu stavové LL-gramatiky.

Šestá kapitola, která se věnuje implementaci, vysvětluje a formálně definuje pojem hierarchické hluboké syntaktické analýzy (dále také HSA). Zejména v této části bude popsáno jak tento *systém* byl použit, respektive, jak usnadnil implementační část této práce. Zároveň zde bude rozebráno několik implementačně důležitých oblastí, jako je například zpracování vstupu, zotavení z chyb či princip a struktura samotné hluboké syntaktické analýzy.

V sedmé kapitole bude představeno několik syntaktických konstrukcí, které klasická, bezkontextová, syntaktická analýza není schopna pokrýt, a které je možno analyzovat pomocí syntaktického analyzátoru implementovaného jako součást této diplomové práce.

Konečně, v osmé kapitole věnované dalším možným směrům výzkumu a vývoje hluboké syntaktické analýzy, bude stručně uvedeno několik málo nejperspektivnějších oblastí, na které již v této práci nezbylo místo nebo jejich složitost značně přesahuje rozsah diplomové práce. Zejména se jedná o rozšíření definice hlubokého ZA o podporu ε-pravidel a představení hypotézy takzvaných makro neterminálních symbolů.

Kapitola 2

Definice a použitá notace

Tato kapitola formálně definuje pojmy, které v této práci budou používány. Některé z těchto pojmů jsou dobře známé a jsou zde uvedeny z části pro úplnost, z části protože pro ně existuje vícero definicí a z části protože některé definice jsou mírně pozměněné pro účely této práce.

2.1 Definice

Definice 2.1.1. Gramatika je čtveřice (N, T, P, S), kde:

- N je konečná množina neterminálů.
- T je konečná množina vstupní abecedy (terminálů).
- P je konečná množina pravidel tvaru A $\rightarrow$ B, kde B $\in$ (N $\cup$ T)$^+$, A $\in$ N.
- S je počáteční neterminální symbol, tedy S $\in$ N.

Definice 2.1.2. LL-gramatika je čtveřice (N, T, P, S), kde:

- N je konečná množina neterminálů.
- T je konečná množina vstupní abecedy (terminálů).
- P je konečná množina pravidel tvaru A $\rightarrow$ B, kde B $\in$ (N $\cup$ T)$^+$, A $\in$ N. Zároveň platí, že $\forall$a $\in$ T a $\forall$A existuje maximálně jedno pravidlo tvaru $A \rightarrow X_1X_2...X_n \in$ P takové, že $a \in First(X_1X_2...X_n)$.
- S je počáteční symbol, tedy S $\in$ N.

Definice 2.1.3. Stavová gramatika G je pětice (Q, Σ, Γ, P, S), kde:

- Q je konečná množina stavů.
- Σ je konečná množina vstupní abecedy (terminálů).
- Γ je úplná abeceda, kde $\Gamma = \Sigma \cup$ N, kde N je konečná množina neterminálů.
- P $\subseteq$ (Q $\times$ N) $\times$ (Q $\times \Gamma^+$) je konečná relace, přičemž místo $(q, A, p, v) \in P$ budeme psát $(q, A) \Rightarrow (p, v) \in P$, případně $qA \rightarrow pv$.

- S je počáteční symbol, tedy S ∈ N.

Povšimněme si, že definice stavové gramatiky zahrnuje pouze počáteční neterminál S. Takto ji definoval Kasai T. s tím, že jako startovní stav se mohou použít všechny takové stavy q, pro které bude existovat pravidlo ve tvaru $qS \rightarrow pv$. Nebo-li, mějme množinu $Q_{start} = \{q \in Q | qS \rightarrow pv \in P$, kde $p \in Q, S$ je startovní neterminál a $v \in \Gamma^+\}$, pak startovní stav $q \in Q_{start}$. Zároveň však pro naše účely tuto vlastnost stavových gramatik můžeme ignorovat, protože my si vždy startovní stav definujeme.

Definice 2.1.4. Derivační krok stavové gramatiky pomocí pravidla $r \in P$ značíme $(q, A) \Rightarrow (p, X)[r]$, kde [r] obvykle vypustíme pro zjednodušení zápisu.

Definice 2.1.5. Stavová gramatika je *n-limited*, pokud každý derivační krok $(q, xAy) \Rightarrow (p, xBy)[(q, A) \rightarrow (p, B)]$ $(p, q \in Q, x, y \in \Gamma^*, B \in \Gamma^+$ a $A \in N)$ splňuje podmínku $|xA|^N \leq n$, kde $|z|^N$ značí počet výskytů neterminálů v řetězci z.

Definice 2.1.6. Zásobníkový automat Z je sedmice (Q, Σ, Γ, R, s, S, F), kde:

- Q je konečná množina stavů, nicméně pro LL-analýzu, která je založena na LL-gramatice se uvažuje pouze jeden, tak jak bylo ukázáno v příkladě číslo 4.22, ve studijní opoře předmětu Teoretická Informatika [3]. Tedy |Q| = 1. Takový ZA pak zřejmě skončí svoji činnost vyprázdněním svého zásobníku.
- Σ je konečná množina vstupní abecedy (terminálů).
- Γ je konečná množina zásobníkové abecedy, kde Γ = Σ ∪ N ∪ {#}, kde N je konečná množina neterminálů a # značí dno zásobníku.
- R je konečná množina pravidel tvaru pAa → qw, kde p, q ∈ Q, A ∈ N, a ∈ Σ a w ∈ Γ*.
- s je počáteční stav.
- S je počáteční symbol na zásobníku.
- F je množina koncových stavů.

Definice 2.1.7. Konfigurace ZA je trojice $Q \times \Sigma^* \times (\Gamma - \{\#\})^*\{\#\}$.

Definice 2.1.8. Výpočetní krok ZA je přechod mezi jeho dvěma konfiguracemi $\alpha = (p, c\gamma, dX)$ a $\beta = (q, e\delta, fZ)$ pomocí nějakého pravidla $r \in R$ značeno $\alpha \Rightarrow \beta$ neboli $(p, c\gamma, dX) \Rightarrow (q, e\delta, fZ)$. Obecně u ZA používaných pro analýzu shora-dolů (LL) můžeme výpočetní krok rozdělit na dva typy, kterými jsou *compare*, který se provede pokud $c = d \wedge c \in \Sigma$, značeno $(p, c\gamma, dX) \Rightarrow^{compare} (q, \delta, Z)$. Toto rozšíříme o možnost několikanásobného *compare*, tedy pokud chceme použít *compare* například čtyřikrát za sebou pro odstranění čtyři symbolů, píšeme: $\alpha \Rightarrow^{compare(4)} \beta$. Druhý typem přechodu je *expansion*, jenž se provede pokud $d \in N$, značeno $(p, c\gamma, dX) \Rightarrow^{expansion} (q, c\gamma, fX)$, kde neterminál d byl expandován na řetězec $f \in (\Gamma - \{\#\})^*$.

Definice 2.1.9. Hluboký zásobníkový automat H je sedmice (Q, Σ, Γ, R, s, S, F), kde:

- Q je konečná množina stavů.
- Σ a Γ jsou definovány stejně jako u ZA.

- R je konečná množina pravidel tvaru npAa $\rightarrow$ qw, kde n $\in \mathbb{N}$, p, q $\in$ Q, A $\in$ N, a $\in \Sigma$ a w $\in \Gamma^*$, a kde n reprezentuje zásobníkovou hloubku ve které je možné toto pravidlo použít.

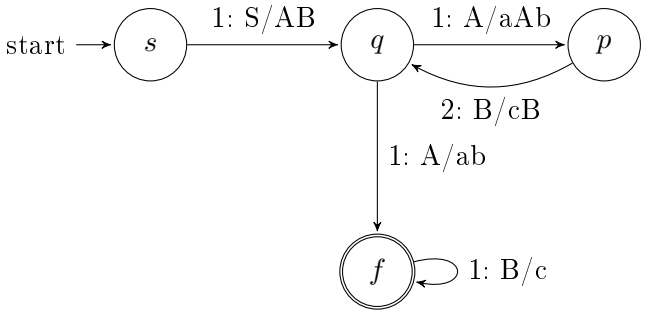
- s, S a F jsou definovány stejně jako u ZA.

Hluboký zásobníkový automat funguje naprosto stejně jako klasický zásobníkový automat pouze s tím rozdílem, že operace *expansion* muže být provedena nejenom pro neterminály na samém vrcholu zásobníku. Mějme dvě konfigurace HZA $\alpha = (p, \gamma, X)$ a $\beta = (q, \delta, Z)$. Expanzi v hlubokém zásobníkovém automatu v hloubce $n \in \mathbb{N}$ budeme značit $\alpha \Rightarrow^{expansion(n)} \beta$ neboli $(p, \gamma, X) \Rightarrow^{expansion(n)} (q, \gamma, Z)$.

Pro upřesnění funkčnosti uvažujme známý kontextový jazyk L = { $a^n b^n c^n$ | n $\geq$ 1 }. Je zřejmé, a pomocí pumping lemma pro bezkotextové jazyky snadno dokazatelné, že takovýto jazyk není možno přijat pomocí klasického zasobníkového automatu. Toto však neplatí pro stavovou gramatiku a hluboký zásobníkový automat.

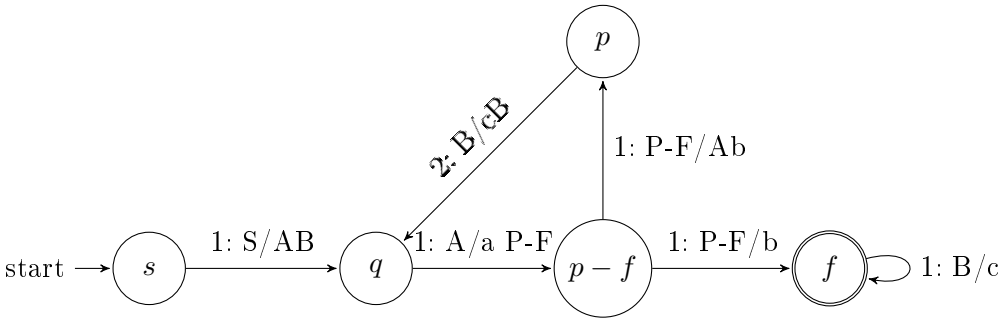
Příklad 2.1.1. Mějme stavovou gramatiku G = {{s, q, p, f}, {a, b, c}, Γ = Q $\cup$ N, P, S}. Dále mějme hluboký zásobníkový automat Z založený na gramatice G. Z = {{s, q, p, f}, {a, b, c}, Γ = Q $\cup$ N, R, s, S, {f}}, kde N = {S, A, B}, přičemž množiny pravidel P a R jsou definovány následovně:

1)	sS	$\rightarrow$	qAB	1)	1sS	$\rightarrow$	qAB
2)	qA	$\rightarrow$	paAb	2)	1qA	$\rightarrow$	paAb
3)	qA	$\rightarrow$	fab	3)	1qA	$\rightarrow$	fab
4)	pB	$\rightarrow$	qcB	4)	2pB	$\rightarrow$	qcB
5)	fB	$\rightarrow$	fc	5)	1fB	$\rightarrow$	fc



Obrázek 2.1: Znázornění automatu Z, respektive gramatiky G, pokud vypustíme čísla hloubek jednotlivých přechodů.

Za povšimnutí stojí, že z definice Z je zřejmé, že gramatika G je 2-*limited*. Tedy, každý derivační krok gramatiky G či automatu Z bude modifikovat maximálně druhý neterminál na zásobníku. Konkrétně automat Z může pracovat následovně:



Obrázek 2.2: Opravený automat Z pomocí faktorizace.

$$
\begin{aligned}
(\text{s, aabbcc, S\#}) &\Rightarrow^{expansion(1)} (\text{q, aabbcc, AB\#})\\
&\Rightarrow^{expansion(1)} (\text{p, aabbcc, aAbB\#})\\
&\Rightarrow^{compare(1)} (\text{p, abbcc, AbB\#})\\
&\Rightarrow^{expansion(2)} (\text{q, abbcc, AbcB\#})\\
&\Rightarrow^{expansion(1)} (\text{f, abbcc, abbcB\#})\\
&\Rightarrow^{compare(4)} (\text{f, c, B\#})\\
&\Rightarrow^{expansion(1)} (\text{f, c, c\#})\\
&\Rightarrow^{compare(1)} (\text{f, } \varepsilon\text{, \#})
\end{aligned}
$$

V prvních dvou výpočetních krocích hluboký ZA pracuje stejně jako klasický ZA, tedy expanduje nejvyšší symbol (neterminál) na zásobníku. V třetím kroce provede porovnání a odstranění terminálního symbolu a jak ze vstupu, tak i ze zásobníku a skončí ve stavu p. Nyní je však nejvrchnějším symbolem na zásobníku neterminál pro nějž ve stavu p neexistuje žádné pravidlo. Proto se hluboký ZA musí podívat hlouběji, konkrétně na neterminál B na druhé pozici v zásobníku (terminály se nepočítají), který pomocí 4. námi definovaného pravidla expanduje. Dále automat bude pracovat stejně jako klasický ZA a skončí s prázdným vstupem a pouze # na zásobníku. Všimněme si však, že HZA Z by ve výše uvedené podobě nefungoval, respektive ve druhém a třetím pravidle zde existuje konflikt, kdy není možné deterministicky rozhodnout, které pravidlo z nich použít. Řešením je obdobně jako u klasického ZA použít faktorizaci (vytýkání). Jedno z možných řešení je naznačeno níže, přičemž pro víceznakové názvy stavů budeme používat notaci <stav> a víceznakové terminály a neterminály budou odděleny mezerou.

$$
\left.\begin{array}{lll} 1qA & \rightarrow & paAb \\ 1qA & \rightarrow & fab \end{array}\right\}
\quad
\begin{array}{lll} 1qA & \rightarrow & \text{<p/f>a P-F} \\ 1\text{<p-f>P-F} & \rightarrow & pAb \\ 1\text{<p-f>P-F} & \rightarrow & fb \end{array}
$$

Ve výsledku byla konfliktní pravidla nahrazena třemi novými a zároveň jsme přidali jeden nový neterminál P-F a jeden nový stav <p-f>. Výsledný automat je vidět na obrázku 2.2.

Kapitola 3

Hluboká LL-analýza bez ε-pravidel

V této kapitole se podíváme na některá úskalí vyloučení ε-pravidel z návrhu gramatik.

3.1 Vyloučení ε-pravidel

Stavové gramatiky, které byly představeny Takumi Kasaiem [9] neuvažovaly použití ε-pravidel. Stejně tak ani definice hlubokých zásobníkových automatů [6] ε-pravidla nepoužívá, jelikož vychází právě ze zmíněné definice stavových gramatik. A víceméně stejně se zachováme i v této práci, přičemž si musíme dát pozor několik problémů a nepříjemností, které tato volba přináší, a které se týkají především návrhu stavové gramatiky.

3.1.1 Faktorizace

V příkladě 2.1.1 jsme si naznačili způsob jak je možné odstranit konflikty ve stavové gramatice, nicméně to nelze provést vždy. Uvažujme například následující pravidla:

$$\begin{aligned} &\text{pA} \rightarrow \text{pa} \\ &\text{pA} \rightarrow \text{qab} \end{aligned}$$

Tato dvě pravidla nelze úspěšně opravit, protože vytkneme-li společný prefix jejich expanzí, tedy řetězec "a" (a vytvoříme-li nezbytný nový stav a neterminál), nutně nám musí vniknout ε-pravidlo:

$$\begin{aligned} &\text{pA} \rightarrow \text{<p-q>a P-Q} \\ &\text{<p-q>P-Q} \rightarrow \text{p}\varepsilon \\ &\text{<p-q>P-Q} \rightarrow \text{qb} \end{aligned}$$

3.1.2 Problematická konstrukce gramatiky

Dalším neduhem nemožnosti použít ε-pravidla je samotný návrh gramatiky. Uvažujme klasickou bezkontextovou LL-gramatiku G = ({PROG, STATS, ST1, ST2}, {'}', '{', ;}, P, PROG), kde P je definováno následovně:

$$\begin{aligned} &\text{PROG} \rightarrow \{\ \text{STATS}\ \} \\ &\text{STATS} \rightarrow \text{ST1}\ ;\ \text{STATS} \mid \text{ST2}\ ;\ \text{STATS} \mid \varepsilon \\ &\text{ST1} \rightarrow \ldots \\ &\text{ST2} \rightarrow \ldots \end{aligned}$$

Idea skrytá za touto gramatikou je zřejmá. Na začátku si vygeneruji hlavní obálku programu a poté generuji jednotlivé příkazy. Nás však zajímá generování neterminálu STATS sebou samým, který je kdykoliv možno smazat pomocí pravidla STATS $\rightarrow \varepsilon$. Bez možnosti použít ε-pravidla se nikdy nesmí vygenerovat neterminál, který se nikdy nepoužije. Tedy, v tomto případě by se z dané gramatiky dalo výše zmíněné pravidlo odstranit například přidáním neterminálu STAT.

PROG $\rightarrow$ { STATS
STATS $\rightarrow$ STAT | }
STAT $\rightarrow$ ST1 ; STATS | ST2 ; STATS
ST1 $\rightarrow$ ...
ST2 $\rightarrow$ ...

Vidíme, že se nám gramatika zvětšila a její čitelnost se snížila. To by samo o sobě až tak nevadilo, nicméně u větších - ale přesto stále relativně malých - gramatik to dělá gramatiku velice špatně čitelnou. Mějme zjednodušený kontextový jazyk L pro definování funkce s libovolným počtem návratových hodnot, 0 parametry ([ ]) a pouze dvěma různými příkazy jenž se mohou objevit v jejím těle:

$$L_{fce} = \{t^n i[\,]\{(ri^n)^?(e(ri^n)^?)^k\}|n,k \geq 1\}.$$

Terminál 't', představuje symbol typu, 'i' představuje identifikátor, 'r' příkaz návratu z funkce, 'e' prázdný příkaz a výraz $x^?$ znamená, že se symbol x (terminál) může opakovat 0 nebo 1, nebo-li $x^? = (x + \varepsilon)$. Tělo funkce lze opustit hned na začátku a pak vždy po libovolně dlouhé sekvenci prázdných příkazů. Například funkce $tti[\,]\{eerii\} \in L_{fce}$ provede dva prázdné příkazy a vrátí dvě proměnné tak jak jí předepisuje její hlavička. Jelikož je jazyk L_{fce} kontextový, může syntaktický analyzátor který jej realizuje, pracovat tak, že si při analýze hlavičky funkce na zásobník poznačí kolik typů má vrátit. Když poté dorazí k $rii\}$ bude vědět kolik jich má vrátit a jelikož pak přímo následuje konec funkce, skončí.

Naproti tomu funkce $tti[\,]\{riieee\} \in L_{fce}$ při kontrole rii způsobí nutnost replikace neterminálů indikujících počet návratových hodnot pro případ, že za bezprostředně následující sekvencí eee, bude další příkaz návratu z funkce. Tady nastává problém protože nemůžeme používat ε-pravidla, která by smazání značek umožnila. Jedním z teoreticky možných způsobů řešení by bylo načíst tokeny až po konec funkce a generovat značky jen tehdy pokud příkaz návratu z funkce najdeme. Nicméně to nemusí být nikterak krátký úsek.

Kapitola 4

Návrh LL-analýzy pro hluboký ZA

Tato kapitola se zabývá problematikou rozšíření standardní bezkontextové LL-analýzy o možnost analyzovat syntaktické konstrukce náležících do třídy kontextových jazyků. Krok po kroku zde budou popsány jednotlivé problémy, které při této snaze mohou nastat.

4.1 LL-tabulka pro hlubokou LL-analýzu

Zřejmým rozdílem mezi syntaktickou LL analýzou založenou na ZA a HZA je podoba LL tabulky. Zatímco v prvním případě je vhodné pravidlo vyhledáváno pomocí kombinace nejvrchnějšího neterminálu na zásobníku a prvního symbolu na vstupu, tak jak je to znázorněno v tabulce 4.1. U hlubokého ZA však k těmto dvou parametrům přibude ještě hloubka pravidla a stav. A protože tyto údaje jsou u HZA přímou součástí levé strany pravidel, logicky se nabízí nahradit neterminál jako klíč v řádku za složený klíč. Tedy (n, s, A), kde n $\in \mathbb{N}$, s $\in$ Q a A $\in$ N. Teoretická podoba této tabulky je opět znázorněna v tabulce 4.2.

	a_1	a_2	$\cdots$	a_i
A_1	$\alpha(A_1, a_1)$	$\alpha(A_1, a_2)$	$\cdots$	$\alpha(A_1, a_i)$
A_2	$\alpha(A_2, a_1)$	$\alpha(A_2, a_2)$	$\cdots$	$\alpha(A_2, a_i)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
A_j	$\alpha(A_j, a_1)$	$\alpha(A_j, a_2)$	$\cdots$	$\alpha(A_j, a_i)$

Tabulka 4.1: Teoretická podoba klasické LL tabulky, kde $\forall A_j \in N$ a $\forall a_i \in \Sigma$.

	a_1	a_2	$\cdots$	a_m
(n_1, s_1, S_1)	$\alpha((n_1, s_1, S_1), a_1)$	$\alpha((n_1, s_1, S_1), a_2)$	$\cdots$	$\alpha((n_1, s_1, S_1), a_m)$
(n_1, s_1, S_2)	$\alpha((n_1, s_1, S_2), a_1)$	$\alpha((n_1, s_1, S_2), a_2)$	$\cdots$	$\alpha((n_1, s_1, S_2), a_m)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
(n_1, s_2, S_1)	$\alpha((n_1, s_2, S_1), a_1)$	$\alpha((n_1, s_2, S_1), a_2)$	$\cdots$	$\alpha((n_1, s_2, S_1), a_m)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
(n_i, s_j, S_k)	$\alpha((n_i, s_j, S_k), a_1)$	$\alpha((n_i, s_j, S_k), a_2)$	$\cdots$	$\alpha((n_i, s_j, S_k), a_m)$

Tabulka 4.2: Teoretická podoba LL tabulky pro HZA, kde $\forall n_i \in \mathbb{N}$, $\forall s_j \in Q$, $\forall S_k \in N$, i, j, k, m ≥ 1 a $\forall a_m \in \Sigma$.

4.2 Hloubka pravidel

V příkladě 2.1.1 jsme si ukázali jednoduchou gramatiku pro jazyk L = { $a^n b^n c^n$ | n $\geq$ 1 }, nicméně už jsme si neukázali jak je tyto hloubky možno získat. Zde se nabízejí dvě možnosti. Jelikož každá stavová gramatika je zároveň *n-limited* stavová gramatika, první možnost je hloubku zcela ignorovat a během samotné syntaktické analýzy jednoduše zkoušet všechny kombinace (d, s, z), kde $0 < d \leq n$, s je aktuální stav a z je aktuální zásobník. A protože je dobrá šance, že n bude poměrně nízké, nemusí to znamenat příliš velké zpomalení.

Druhou možností je použít algoritmus 4.2.1, jenž funguje na principu prohledávání stavového prostoru do šířky. Ze startovního stavu se použijí všechna možná pravidla, přičemž se bude hloubka použitých pravidel zaznamenávat. Taktéž se budou poznamenávat všechna použitá pravidla pro každou jednu prohledávanou větev samostatně a jakmile by se mělo použít nějaké pravidlo podruhé, daná větev je zahozena.

Algoritmus 4.2.1. Výpočet množin hloubek pravidel stavové gramatiky.

Vstup: Stavová gramatika G = (Q, Σ, Γ, P, S) a počáteční stav s.
Výstup: *depths* = { (r, d_r) | $r \in P$, kde $\forall h \in d_r : h \geq 1 \wedge d_r$ je množina všech hloubek pravidla r. }

1. Mějme množinu $depths = P \times \{\{\}\}$ a seznam trojic (stav, zásobník, použitá_pravidla) $open = [(s, S, \{\})]$.

2. Vyjmu první trojici $(state, stack, set)$ ze seznamu *open* a provedu jeden derivační (*expansion*) krok na nejvyšší možné pozici zásobníku *stack* ze stavu *state* pomocí $\forall r \in P$: $r \notin set$. Formálně tedy provedeme expanzi pomocí všech pravidel $r : pA \to qv$ takových, že $p = state$ a A je nejvyšší možný neterminál na *stack* takový, že existuje pravidlo $r \in P$ s levou stranou pA. Pokud žádné takové pravidlo r neexistuje nebo pokud pro provedení pravidla není k dispozici potřebný neterminál A, vyjmutý záznam zahodím.

3. Pro všechny získané stavy vytvořím nové trojice $(q, newStack, set \cup \{r\})$, kde $newStack$ je zásobník získaný jednou expanzí (*expansion*) pomocí pravidla r a následným odstraněním všech terminálů z vrcholu zásobníku. Všechny nové trojice vložím na konec seznamu *open*.

4. Hloubku neterminálu A, který pravidlo r expandovalo na řetězec v, poznamenám do množiny *depths* k příslušnému pravidlu, tedy r.

5. Opakuji od 2. bodu dokud seznam *open* není prázdný.

Příklad 4.2.1. Mějme relaci P definovanou následovně:

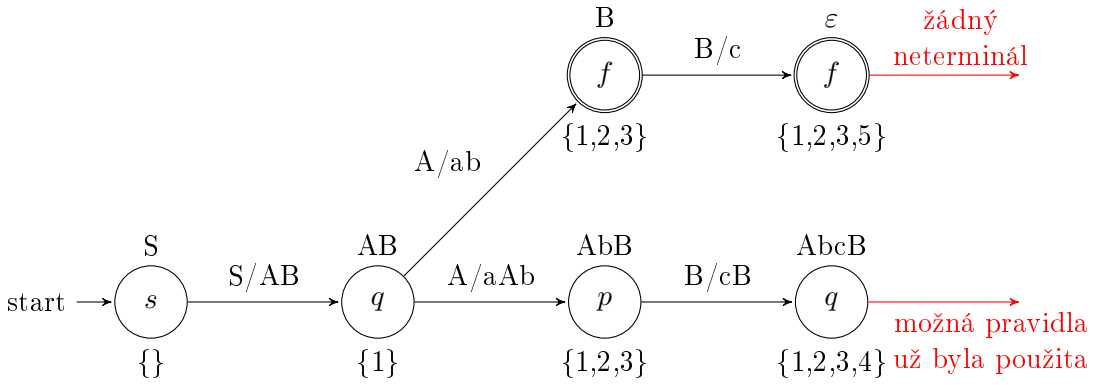
1:	sS	→	qAB
2:	qA	→	paAb
3:	qA	→	fab
4:	pB	→	qcB
5:	fB	→	fc

Dále mějme seznam $open = [(s, S, \{\})]$, množinu $depths = P \times \{\{\}\} \Rightarrow depths = \{(sS \to qAB, \{\}), (qA \to paAb, \{\}), (qA \to fab, \{\}), (pB \to qcB, \{\}), (f \to fc, \{\})\}$. Pro přehlednost místo celých pravidel budeme v následujícím příkladu používat pouze jejich číselné označení uvedené výše. Opakovaným prováděním kroků 2-5 pak dostaneme:

krok	*open*
0.	$[(s, S, \{\})]$
1.	$[(q, AB, \{1\})]$
2.	$[(p, AbB, \{1,2,3\}), (f, B, \{1,2,3\})]$
3.	$[(f, B, \{1,2,3\}), (q, AbcB, \{1,2,3,4\})]$
4.	$[(q, AbcB, \{1,2,3,4\}), (f, \varepsilon, \{1,2,3,5\})]$
5.	$[\,]$

krok	*depths*
0.	$\{(1,\{\}), (2,\{\}), (3,\{\}), (4,\{\}), (5,\{\})\}$
1.	$\{(1,\{1\}), (2,\{\}), (3,\{\}), (4,\{\}), (5,\{\})\}$
2.	$\{(1,\{1\}), (2,\{1\}), (3,\{1\}), (4,\{\}), (5,\{\})\}$
3.	$\{(1,\{1\}), (2,\{1\}), (3,\{1\}), (4,\{2\}), (5,\{\})\}$
4.	$\{(1,\{1\}), (2,\{1\}), (3,\{1\}), (4,\{2\}), (5,\{1\})\}$
5.	$\{(1,\{1\}), (2,\{1\}), (3,\{1\}), (4,\{2\}), (5,\{1\})\}$

Z množiny *depths* kroku 5 je zřejmé, že všechna pravidla kromě 4. jsou použitelná v hloubce 1. Čtvrté pravidlo naopak v hloubce 2. Také si všimněme červeně zvýrazněných záznamů v seznamu *open*, ze kterých není možné nic expandovat a proto je seznam *open* v pátém kroku prázdný. V případě druhé položky třetího kroku a první položky čtvrtého kroku by bylo možné použít jedině pravidla 2 a 3, ale ty už máme označená jako použitá. V druhém záznamu naopak není žádný neterminál k expanzi. Pro zřetelnost obrázek 4.1 znázorňuje prohledávání stavového prostoru. Nad každým stavem je uveden aktuální obsah zásobníku (prefix terminálů se odstraňuje), zatímco pod stavem je množina použitých pravidel.



Obrázek 4.1: Grafické znázornění algoritmu 4.2.1.

Algoritmus 4.2.1 funguje pouze relativně dobře, protože nemá zaručeno, že nalezne všechny možné hloubky. Uvažujme tato pravidla:

$$sS \to qAAB \qquad qA \to qa \qquad qB \to pXX$$

Je zřejmé, že pro přechod ze stavu s do stavu p, je nutné dvakrát uplatnit pravidlo $qA \to qa$ ve stavu q, což algoritmus 4.2.1 neumožňuje. Proto je téměř nutné algoritmus modifikovat tak, aby bylo možné vícenásobné použití pravidla v jednom stavu. Zde se však musí

postupovat opatrně, protože každé navýšení limitu opakování o 1 způsobí značný nárůst stavového prostoru a to i pro malé stavové gramatiky, protože časová složitost je exponenciální. Stanoví-li se tedy například limit na dvě, pak každé pravidlo s počátečním stavem q musí být v navrhované stavové gramatice proveditelné maximálně po dvojnásobném použití všech ostatních pravidel s počátečním stavem q. V opačném případě takové pravidlo bude nedosažitelné.

Doplňme ještě, že oba představené způsoby nalezení (nenalezení) hloubek jednotlivých pravidel nejsou sama o sobě dostatečná pro realizaci hluboké syntaktické LL-analýzy. Důvod a řešení bude vysvětleno v podkapitole 4.5.

4.3 Množina First

Pro samotnou konstrukci LL-tabulky pro klasickou LL-analýzu jsou potřeba množiny *Empty*, *First*, *Follow* a *Predict*, z nichž všechny kromě množiny *First* byly přidány především z důvodu zahrnutí ε-pravidel, tedy je můžeme pro holou hlubokou SA ignorovat. Na druhou stranu množinu *Follow* budeme potřebovat v kapitole rozebírající zotavení z chyb.

Množina *First* je v klasické LL-analýze definována dle vztahu 4.3.1.

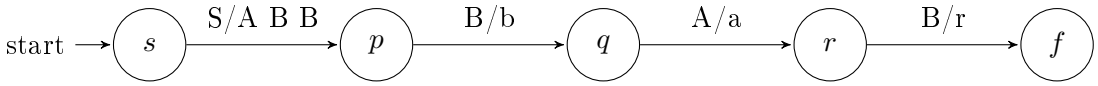
$$First(X) = \begin{cases} \{\mathrm{X}\} & X \in \Sigma \\ \{\mathrm{x} \in \Sigma | x \text{ může být 1. v sekvenci derivací } X\} & X \in N. \end{cases} \tag{4.3.1}$$

Naproti tomu množina *First* pro hlubokou LL-analýzu (dále jen $First_d$) bude odrážet změny LL-tabulky navržené v sekci 4.1. Mějme *n-limited* stavovou gramatikou G_n, množinu $D = \{1, 2, ..., n\}$ všech možných hloubek pravidel gramatiky G_n. Dále definujme množinu $KEY_G = D \times Q \times N$ reprezentující všechny možné klíče hluboké LL-tabulky, pak funkce množiny $First_d$ je definována dle vztahu 4.3.2.

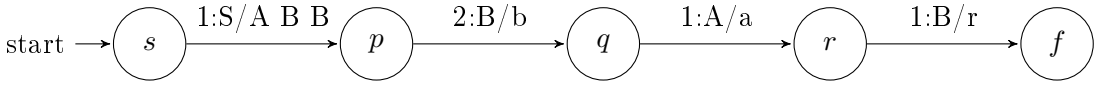
$$First_d(X) = \begin{cases} \{\mathrm{X}\} & X \in \Sigma \\ \{\mathrm{x} \in \Sigma | \exists r : npA \to qv \in P : \mathrm{x} \text{ může být 1. v sekvenci derivací z } npA\ \} & X \in KEY_G. \end{cases} \tag{4.3.2}$$

Algoritmus 4.3.1 formálně znázorňuje výpočet množiny *First* (pro gramatiku bez ε-pravidel). Naproti tomu, pro výpočet množiny $First_d$ se nejdříve musíme vypořádat s faktem, že řádkovým klíčem hluboké LL-tabulky je kromě neterminálu i stav. To znamená, že s přechodem mezi stavy se samotný klíč může vlivem přítomnosti pravidel operujících mimo vrchol zásobníku, měnit. Jako příklad uvažujme derivační krok stavové gramatiky $pABC \Rightarrow qAbC$, kde neterminál B byl přepsán na terminál b. Je vidět že klíč (část klíče, uvažujeme-li pravidlo pro hluboký ZA - ignorujeme hloubku) pA se změnil na qA a striktně řečeno, pro porovnání jsou oba dva zcela odlišné. Jako řešení se nabízí využít takzvaného uzávěru stavu (tranzitivního), což je množina všech stavů do kterých se při využití libovolných pravidel stavové gramatiky můžeme dostat, čímž v podstatě vytvoříme jakýsi slovník *synonym* stavů. Uvažme výše uvedený příklad, kde z derivace $pABC \Rightarrow qAbC$ jasně plyne, že musí existovat pravidlo $2pB \to qb$ a tedy množina StateClosure(p) vypočtená podle algoritmu 4.3.2 bude rovna {p,q}.

Dále, jak je to s hloubkou pravidel stavové gramatiky? Uvažujme stavovou gramatiku $\mathrm{G}_{ex1} = (\{s, p, q, r, f\}, \{a, b, r\}, \{a, b, r, S, A, B\}, P, S)$ na obrázku 4.2. Je zřejmé, že jazyk $L(G_{ex1})$ obsahuje jediný řetězec abr. Pokud bychom ignorovali hloubky pravidel (obrázek



Obrázek 4.2: Stavová gramatika G_{ex1}.



Obrázek 4.3: Stavová gramatika G_{ex1} doplněná o hloubky pravidel dle algoritmu 4.2.1.

4.3) nutně by to za použití množiny $StateClosure(p)$ znamenalo, že $First_d(pB) = \{b, r\}$, což je špatně, protože pak by hluboký ZA akceptoval $L(G_{ex1}) = \{abr, arr\}$.

Konečně, když známe postup výpočtu množin $First$ a $StateClosure$, můžeme formálně definovat algoritmus 4.3.3 pro výpočet množiny $First_d$.

Algoritmus 4.3.1. Výpočet množiny $First$.
Vstup: LL gramatika G = (N, T, P, S) bez ε-pravidel.
Výstup: $First(X)$ pro každé $X \in N \cup T$.

1. $\forall a \in T : First(a) = \{a\}$.
2. $\forall r \in P : A \to aX, A \in N, a \in T, X \in (T \cup N)^*, : First(A) = \{a\}$.
3. $\forall r \in P : A \to BX; A, B \in N, X \in (T \cup N)^*, :$ Přidej $First(B)$ do $First(A)$ a opakuj toto pravidlo dokud se alespoň jedna množina $First$ mění.

Algoritmus 4.3.2. Výpočet množiny $StateClosure$.
Vstup: Stavová gramatika G = (Q, Σ, Γ, P, S) bez ε-pravidel.
Výstup: $StateClosure(q)$ pro každé $q \in Q$.

1. $\forall p \in Q : StateClosure(p) = \{p\}$.
2. $\forall p, q \in Q : \exists r \in P : npA \to qB$, kde $B \in \Gamma^+ : StateClosure(p) = StateClosure(p) \cup StateClosure(q)$.
3. Opaku pravidlo 2 dokud se alespoň jedna množina $StateClosure$ mění.

Algoritmus 4.3.3. Výpočet množiny $First_d$.
Vstup: Stavová gramatika G = (Q, Σ, Γ, P, S) bez ε-pravidel a doplněná o hloubky pravidel pomocí algoritmu 4.2.1.
Výstup: $First_d(X)$ pro každé $X \in \Sigma$ a pro každé $X = npA$.

1. $\forall a \in \Sigma : First_d(a) = \{a\}$.
2. $\forall r \in P : npA \to qaX; n \in \mathbb{N}; p, q \in Q, A \in N, a \in \Sigma, X \in \Gamma^* : First_d(npA) = First_d(npA) \cup \{a\}$.

3. $\forall r \in P$, kde $r : npA \rightarrow qBX : n \in \mathbb{N}; p, q \in Q; A, B \in N; X \in \Gamma^*$ a $\forall s \in StateClosure(q)$ přidej $First_d(nsA)$ do $First_d(npA)$ a opakuj toto pravidlo dokud se alespoň jedna množina $First_d$ mění.

Uvažujme příklad s gramatikou v tabulce 4.3 a jejími *StateClosure* množinami, tabulka 4.4. Pak její výsledné množiny $First_d$ jsou v tabulce 4.5.

1sS	→	qA B
1qA	→	<p-f>a P-F
1<p-f>P-F	→	pA b
1<p-f>P-F	→	fb
2pB	→	qc B
1fB	→	fc

Tabulka 4.3: Stavová gramatika z obrázku 2.2

StateClosure(s)	{s, q,p-f, p, f}
StateClosure(q)	{q, p-f, p, f}
StateClosure(p-f)	{q, p-f, p, f}
StateClosure(p)	{q, p-f, p, f}
StateClosure(f)	{f}

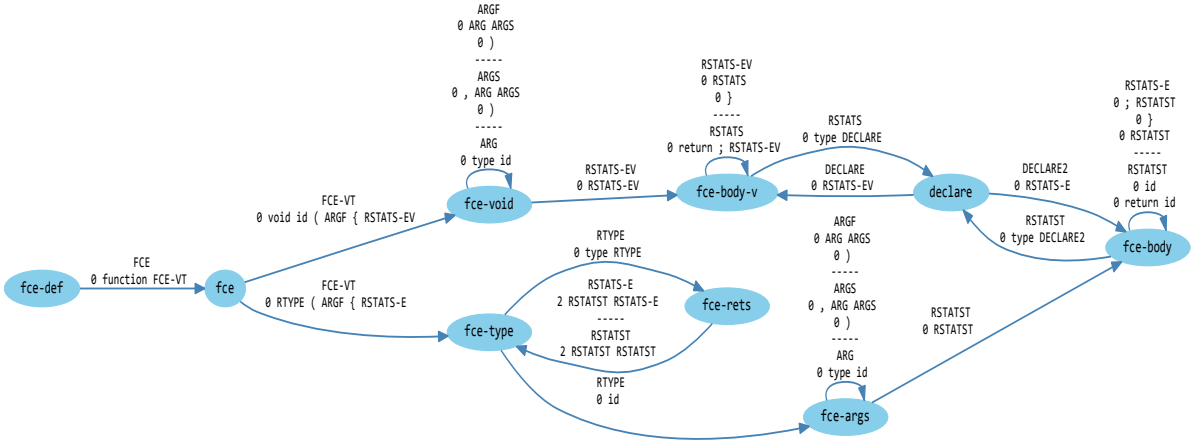
Tabulka 4.4: Znázorňující množiny $StateClosure(s)$ $\forall s \in Q$.

$First_d(1sS)$	{a}
$First_d(1qA)$	{a}
$First_d(1\langle p\text{-}f\rangle P\text{-}F)$	{a, b}
$First_d(2pB)$	{c}
$First_d(1fB)$	{c}

Tabulka 4.5: Výsledné množiny $First_d$.

Za zmínku stojí, že algoritmus pro výpočet uzávěru stavů by mohl být pro účely hluboké syntaktické analýzy přesnější. Uvažujme stavovou gramatiku na obrázku 4.4. Pokud bychom užili algoritmus 4.3.2 pro výpočet dostupných stavů ze stavu *fce-type*, dostaneme $\{fce\text{-}type, fce\text{-}rets, fce\text{-}args, fce\text{-}body, declare, fce\text{-}body\text{-}v\}$. Poslední stav *fce-body-v* je zahrnut chybně, jelikož pro přechod ze stavu *declare* do stavu *fce-body-v* nutně potřebujeme neterminál DECLARE, který však nikdy nezískáme. Naopak, vždy budeme mít neterminál DECLARE2, který povoluje pouze návrat do stavu *fce-body*. Jako řešení se nabízí opět použít prohledávání stavového prostoru do šířky, kdy postupnou simulací SA s limitem opakování pravidel (popsáno v podkapitole 4.2) jsme schopni odhalit všechny opravdu dostupné stavy, což popisuje algoritmus 4.3.4.

Algoritmus 4.3.4. Výpočet množiny *StateClosure* pomocí prohledávání stavového prostoru.
Vstup: Stavová gramatika G = (Q, Σ, Γ, P, S) bez ε-pravidel.
Výstup: $StateClosure(q)$ pro každé $q \in Q$.



Obrázek 4.4: Příklad stavové gramatiky.

1. Mějme množinu uspořádaných seznamů stavů $statePathList = \{\}$ a seznam trojic (stav, zásobník, použitá_pravidla, navštívené_stavy) $open = [(s, S, \{\}, \{s\})]$.

2. Vyjmu první čtveřici $(state, stack, set, visited)$ ze seznamu $open$ a provedu jeden derivační ($expansion$) krok na nejvyšší možné pozici zásobníku $stack$ ze stavu $state$ pomocí $\forall r \in P$: $r \notin set$. Formálně tedy provedeme expanzi pomocí všech pravidel $r : pA \to qv$ takových, že $p = state$ a A je nejvyšší možný neterminál na $stack$ takový, že existuje pravidlo $r \in P$ s levou stranou pA. Pokud žádné takové pravidlo r neexistuje nebo pokud pro provedení pravidla není k dispozici potřebný neterminál A, seznam $visitied$ vložím do množiny $statePathList$ a zbytek vyjmutého záznamu zahodím.

3. Pro všechny získané konfigurace (neformálně míněno) vytvořím nové čtveřice $(q, newStack, set \cup \{r\}, visited + q)$, kde $newStack$ je zásobník získaný jednou expanzí ($expansion$) pomocí pravidla r a následným odstraněním všech terminálů z vrcholu zásobníku. Všechny nové čtveřice vložím na konec seznamu $open$.

4. Opakuj od 2. bodu dokud seznam $open$ není prázdný, pak pokračuj bodem 5.

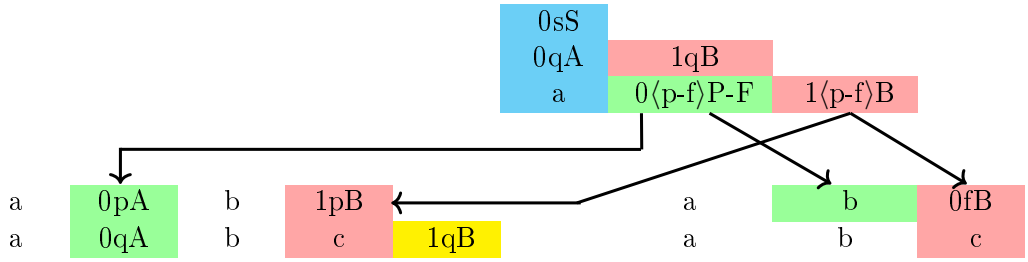
5. Nyní množina $StateClosure(x) = \{q \in Q | \forall list \in statePathList$ a $\forall q \in list : q$ se v seznamu $list$ vyskytuje za stavem $x\}$.

6. Bod 5. provedu $\forall q \in Q$.

Další možností je spočítat množinu $First_d$ přímo, s využitím prohledávání stavového prostoru, jelikož tento postup zahrnuje uzávěry stavů implicitně. Způsob by byl podobný jako v algoritmu 4.3.4, ale místo sekvencí navštívených stavů bychom shromažďovali sekvence klíčů hluboké LL tabulky, přičemž jednotlivé sekvence by byly přesunovány do seznamu sekvencí v případě, že by se je podařilo přepsat na terminál nebo daná sekvence již dále nejde expandovat. Například při dosažení limitu možných opakování pravidla v jednom stavu by byla sekvence uložena pro případ, že některá její položka (klíč) již má, nebo bude mít terminální sekvenci. Blíže znázorněno je to v tabulce 4.5, kde je pro přehlednost ke každému

neterminálu v každé větné formě zopakována hloubka (pozice neterminálu ve větné formě při ignorování terminálních symbolů) a aktuální stav.

Na prvním řádku je neterminál S expandován na A B a dále je pak neterminál A přepsán na a. Z toho je zřejmé, že $\forall a \in First_d(0qA) \rightarrow a \in First_d(0sS)$, což je také zvýrazněno modrou barvou. Nazvěme tuto sekvenci X, pak platí: $\forall k \in X : a \in First_d(k)$.



Obrázek 4.5: Ilustrace výpočtu množiny $First_d$ prohledáváním stavového prostoru.

Druhou sekvencí která se v diagramu vyskytuje (červeně) je [1qB, 1⟨p-f⟩B, [[1pB,c],[0fB,c]]], která je z třetího na čtvrtý řádek rozdvojena, protože zde bylo možné použít dvě různá pravidla. To samé platí o zelené sekvenci, jenž vznikla expandováním neterminálu A na druhém řádku. Zajímavý je případ na posledním řádku, kdy zelená sekvence skončila trojicí 0qA, což však nevadí, protože už je obsažena v modré sekvenci a je tedy ukončena symbolem 'a'. To znamená , že 'a' $\in First_d(0qA)$ a zároveň 'a' $\in First_d(0\langle p\text{-}f\rangle P\text{-}F)$.

Algoritmus 4.3.5 popisuje jak strukturu na obrázku 4.5 sestrojit a jak z ní získat potřebné množiny.

Algoritmus 4.3.5. Výpočet množiny $First_d$ prohledáváním stavového prostoru.
Vstup: Stavová gramatika G = (Q, Σ, Γ, P, S) bez ε-pravidel.
Výstup: $First_d(k)$ pro každý klíč k hluboké LL-tabulky.

1. Simulujme hlubokou SA s limitem opakování pravidel pro jednotlivé stavy podobně jako v algoritmech 4.2.1 a 4.3.4 a zároveň mějme prázdný seznam sekvencí klíčů hluboké LL-tabulky L. Navíc mějme funkci $pos(x)$, která pro hloubku neterminálu x vrátí pozici na zásobníku (zahrne i terminály).

2. Každá prohledávaná větev si udržuje vlastní seznam sekvencí M. M[n] je sekvence v seznamu M na pozici n.

3. Po každé expanzi za pomocí pravidla $dpA \rightarrow qxX$, kde $x \in \Gamma$ a $X \in \Gamma^*$, proveď tyto úkony:

   a) $\forall B \in N$ ve větné formě (obsah zásobníku) přidej trojici nqB do sekvence M[n], kde $0 \leq n < pos(d)$ a je n udává pozici na zásobníku.

   b) Pozice d větné formy. Pokud je prvním symbolem expanze terminál ($x \in \Sigma$), ukonči jím sekvenci v seznamu M[n] a vlož ji do seznamu L. Jinak proveď bod a), kde $n = pos(d)$.

   c) $\forall B \in N$: vytvoř novou sekvenci, vlož do ní trojici nqB a tuto sekvenci vlož do M na pozici n, kde $d < n < d + length(xX)$.

   d) $\forall B \in N$ ve větné formě přidej trojici nqB do sekvence M[n], kde $d+length(xX) \leq n$.

4. Bod 3. opakuj dokud je to možné, pak přesuň všechny sekvence neukončené terminálem do seznamu L.

5. Ze získaného seznamu sekvencí vypočteme množiny $First_d$ následovně:

   a) $\forall s \in L : s$ je ukončeno terminálem, $\forall k \in s$, přidej ukončovací terminál sekvence s do $First_d(k)$.

   b) $\forall s \in L : s$ není ukončeno terminálem, $\forall j, k \in s : k$ je v sekvenci s za j, přidej $First_d(k)$ do $First_d(j)$. (Tento bod je či není potřeba v závislosti na velikosti limitu opakování pravidel ve stavu gramatiky).

6. Bod 5. opakuj dokud se kterákoliv množina $First_d$ mění.

4.4 Množina Follow

Algoritmus pro výpočet množin *Follow* pro hlubokou syntaktickou analýzu (dále označován jako $Follow_d$) je velice podobný jeho klasické variantě, bohužel i on trpí stejnými neduhy jako algoritmus $First_d$, což je zejména proměnná podoba složených klíčů. Proto opět použijeme algoritmus na bázi prohledávání stavového prostoru.

Algoritmus 4.4.1. Výpočet množiny $Follow_d$ prohledáváním stavového prostoru.
Vstup: Stavová gramatika G = (Q, Σ, Γ, P, S) bez ε-pravidel.
Výstup: $Follow_d(k)$ pro každý klíč k hluboké LL-tabulky.

1. Simulujme hlubokou SA s limitem opakování pravidel pro jednotlivé stavy podobně jako v algoritmech 4.2.1 a 4.3.4 a zároveň mějme prázdný seznam L dvojic (a, b), kde a je vždy složený klíč hluboké LL-tabulky a b může být jak složený klíč, tak terminální symbol.

2. Po každé expanzi za pomocí pravidla $dpA \to qxX$, kde $x \in \Gamma$ a $X \in \Gamma^*$, proveď tyto úkony:

   a) Mějme první symbol v nalevo od nově expandované části a symbol w odpovídající prvnímu symbolu v expanzi právě použitého pravidla, tedy x.

   b) Pokud je v terminál, nastav $v = w$, $w =$ další symbol v expanzi xX a opakuj bod b), jinak pokračuj bodem c).

   c) Symbol v je neterminál. Přidej dvojici (v, w) do seznamu L, nastav $v = w$ a $w =$ další symbol v expanzi xX a opakuj bod b).

   d) V expanzi xX není další symbol pro přiřazení do w, pak nastav w na bezprostředně následující symbol za expanzí xX, a pokud v je neterminál, přidej dvojici (v, w) do seznamu L a ukonči bod 2.

3. Bod 2. opakuj po každé expanzi.

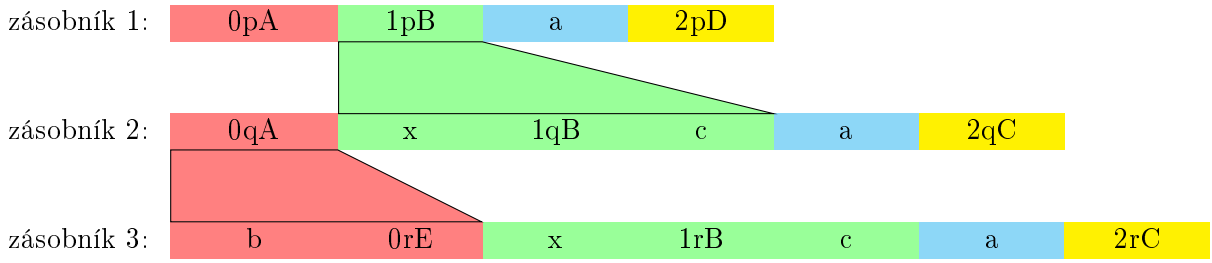
4. Ze získaného seznamu L dvojic (a, b) vypočteme množiny $Follow_d$ následovně:

   a) $\forall (a, b) \in L, b \in \Sigma$: proveď $Follow_d(a) = Follow_d(a) \cup \{b\}$.

   b) $\forall (a, b) \in L, b \in N$: proveď $Follow_d(a) = Follow_d(a) \cup Follow_d(b)$.

5. Bod 4. opakuj dokud se kterákoliv množina $Follow_d$ mění.

Uvažme obrázek 4.6, kde jsou znázorněny dvě expanze v jedné větvi algoritmu prohledávající stavový prostor (do šířky), respektive tři po sobě následující konfigurace hlubokého ZA, přičemž pro přehlednost je ke každému neterminál přidána jeho hloubka a aktuální stav.



Obrázek 4.6: Ilustrace výpočtu množin $Follow_d$.

V příkladě byla použita pravidla $1pB \to qxBc$ a $0qA \to rbE$, přičemž z první konfigurace je dle algoritmu 4.4.1 zřejmé, že získáme pouze dvojice (0pA, 1pB) a (1pB, a). Dvojice (a, 2pD) je nezajímavá. V druhé konfiguraci můžeme ignorovat všechno kromě právě expandované části zásobníku a jeho dvou bezprostředních sousedů (levá a pravá strana). Získáme tedy dvojice (0qA, x) a (1qB, c). V třetí konfiguraci můžeme vypustit všechno kromě tří prvních položek, čímž získáme jedinou dvojici (0rE, x). Zde si povšimněme, že poslední dvojice byla získána pomocí bodu 2.a), algoritmu 4.4.1, který slouží jako náhrada za krok přiřazení expandovaného neterminálu do posledního neterminálu v pravidle, při výpočtu klasické množiny *Follow*. Jinak řečeno, pokud je neterminál na konci expanze nějakého pravidla, tak musí být nutně následován vším, čím je následován expandovaný neterminál.

Celkově tedy získáme seznam L = [...,(0pA, 1pB), (1pB, a), (0qA, x), (1qB, c), (0rE, x),...], ze kterého množiny $Follow_d$ získáme provedením kroků 4 a 5, konkrétně tedy:

$$\begin{aligned}
&\dots \\
Follow_d(0pA) &= Follow_d(0pA) \cup Follow_d(1pB) \\
Follow_d(1pB) &= Follow_d(1pB) \cup \{a\} \\
Follow_d(0qA) &= Follow_d(0qA) \cup \{x\} \\
Follow_d(1qB) &= Follow_d(1qB) \cup \{c\} \\
Follow_d(0rE) &= Follow_d(0rE) \cup \{x\} \\
&\dots
\end{aligned}$$

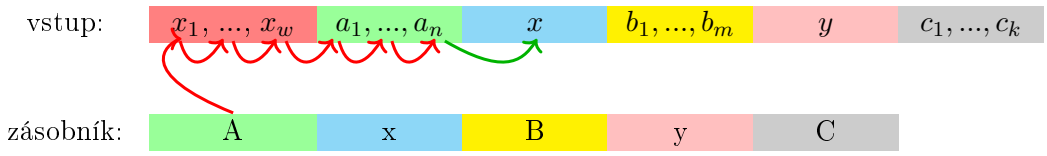
4.5 Srovnání principu zotavení z chyb

Zotavení z chyb není cílem této práce, přesto však rozebereme alespoň základní srovnání zotavení z chyb u klasické a hluboké LL-analýzy a jaké problémy je třeba vyřešit pro jeho dosažení.

Při klasickém zotavení z chyb mohou nastat dvě základní chyby. Prvním z nich je neočekávaný *token* při provádění *compare*. Ta může nastat pokud je ve vstupním řetězci chybná ta část, která je při *compare* srovnávána s terminály expandovanými mimo první pozici pravidla. Například pravidlo $A \to abc$ je uplatnitelné jen pokud na vstupu najdu terminál 'a', znaky 'b' a 'c' jsou však kontrolovány jen pomocí *compare*.

Druhým případem je situace, kdy nemůžeme aplikovat žádné pravidlo. V tom případě je na vrcholu zásobníku neterminál X a na vstupu *token*, který není obsažen ve $First(X)$.

Běžně se tyto situace řeší přeskočením části vstupu až po klíč, což je typicky symbol náležící do $Follow(X)$, což znázorňuje obrázek 4.7. Stejnou barvou jsou zde zvýrazněny ty neterminály a části vstupního řetězce, které si samy sobě odpovídají. Červeně podbarvená je chybná (nadbývající) část vstupu. Pro zjemnění se může do vyhledávání klíče zahrnout i množina $First(X)$, což zamezí přeskočení části $a_1, ..., a_n$.

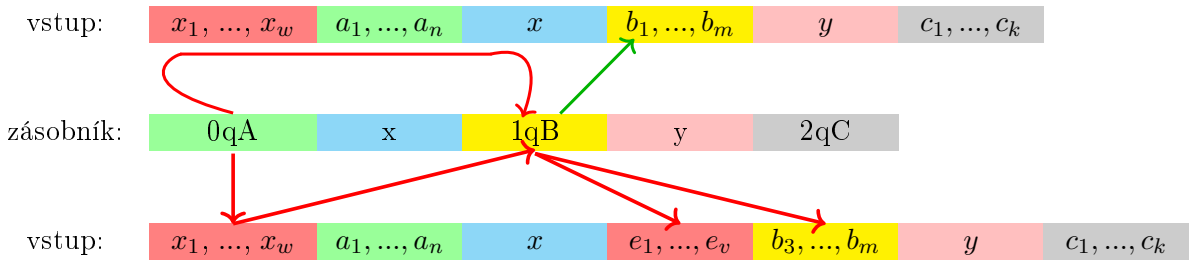


Obrázek 4.7: Ilustrace zotavení z chyb při klasické LL-analýze.

U hluboké syntaktické analýzy je toto chování podobné, nicméně zde vyvstává jeden podstatný rozdíl. Na obrázku 4.8 máme totožnou situaci jako v předchozím případě uzpůsobenou pro hlubokou SA. Přesněji řečeno, aktuální stav je q a na zásobníku máme tři neterminály A, B a C v hloubkách 0,1 a 2. Zároveň uvažujme, že dle definice gramatiky je možno ve stavu q použít dvě pravidla, jejichž levé strany jsou $0qA$ a $1qB$.

Na začátku vstupu máme chybný prefix řetězce $x_1, ..., x_w$, což způsobí, že pravidlo $0qA$, nebude aplikováno a v normálním případě by se zahájila procedura pro zotavení. U hluboké SA však nejdříve vyzkoušíme pravidlo $1qB$, které by si úspěšně přednačetlo vstup a expandovalo by neterminál B na řetězec $b_1, ..., b_m$. To samozřejmě povede na chybu, protože (například) do stavu q, se již proces SA nikdy nemusí vrátit.

K tomu navíc může nastat i případ, kdy vstupní řetězec je natolik poškozen, že ani jedno z použitelných pravidel se nepodaří provést (druhý a třetí řádek obrázku 4.8). Nejprve se vyzkouší zda-li jde aplikovat pravidlo pro vrchol zásobníku, což selže a automaticky se předpokládá, že je potřeba použít pravidlo ve větší hloubce. To selže také, což povede na zotavení z chyb.



Obrázek 4.8: Ilustrace zotavení z chyb při hluboké LL-analýze.

Z předchozích příkladů plyne jeden zásadní poznatek. Kdykoliv je hluboká syntaktická analýza ve stavu q, ve kterém je možné použít alespoň dvě pravidla v různých hloubkách d_1 a d_2 a aplikuje-li se pravidlo v hloubce d_2, kde $d_2 > d_1$, není jasné, zda pravidlo pro hloubku d_1 bylo přeskočeno z důvodů chyby vstupu, nebo protože opravdu mělo být přeskočeno. To je samozřejmě nepřijatelné, protože to znamená, že hluboká SA v takové podobě je nedeterministická.

Jedním intuitivně nabízejícím se řešením je přidat omezení na stavovou gramatiku: $\forall p \in Q, \nexists r_1 : npA_1 \to q_1B_1, r_2 : mpA_2 \to q_2B_2 \in P : m = n$, kde $m, n \geq 0, B_1, B_2 \in \Gamma$. Tedy by mělo platit, že v každém stavu existují pouze pravidla operující nad jednou hloubkou zásobníků.

Naneštěstí tato změna má minimálně jeden nepříjemný důsledek. V podkapitole 3.1.2 bylo stručně popsáno několik problémů vyplývajících z vyloučení ε-pravidel. Tím, který nás momentálně zajímá, je případ přednačítání vstupu, kdy je zapotřebí zkontrolovat jestli vstupní řetězec obsahuje další výskyt nějakého symbolu či ne a jen při jeho výskytu chceme replikovat nějaký obsah na zásobníku. To nadále již není možné vůbec nebo jen ve velice omezené míře, jelikož omezení jedné hloubky na stav gramatiky omezuje generativní sílu stavové gramatiky.

Uvažujme například případ kontextové SA pro kontrolu zda-li se počet argumentů v prototypu funkce a počet argumentů v jejím volání shoduje. V takové situaci není možné použít stejný přístup jako v případě jazyka $L = \{a^n b^n c^n | n \geq 1\}$, kdy se shodný počet znaků 'a', 'b' a 'c' kontroluje paralelně, protože jednotlivé výskyty volání funkce mohou být proloženy libovolně dlouhými sekvencemi jiných příkazů. Jediným řešením je při kontrole prototypu funkce na zásobník vygenerovat jakési poznámky v podobně neterminálů označující správný počet argumentů. Při výskytu volání funkce se však zároveň potřebujeme podívat dále a zjistit jestli je za tímto voláním další, abychom nereplikovali neterminály zbytečně.

Právě tady nastává problém, protože počet parametrů funkce je obecně proměnný a tedy i pravidla snažící se najít další výskyt musí pracovat v proměnné hloubce, což však již není možné. Uvažujme následující obsahy zásobníku:

a) `N N N ... FUNCTION-CALL`
b) `N N N N N ... FUNCTION-CALL`

Je zřejmé, že jakékoliv neterminály vyskytující se v části `FUNCTION-CALL` a kontrolující další výskyt nějakého symbolu či tokenu, musí v případě (a) pracovat v hloubce 4 a výš, respektive v hloubce 6 a výše v případě (b).

Jaká přesně je generativní síla stavové gramatiky s omezením na jednu hloubku na její stav není jasné, nicméně je možné její sílu zařadit alespoň přibližně. Jelikož je stále možné sestavit gramatiku pro vstupní řetězce jako je `int bool string number flag str = 666 false "Hello World!"`, je zřejmé, že její síla je větší než bezkotextové gramatiky, tedy pokud označíme třídu jazyků definovaných stavovými gramatikami obsahující jen pravidla v jedné hloubce na stav $\mathcal{L}_{SG1}$, můžeme formálně zapsat $\mathcal{L}_2 \subset \mathcal{L}_{SG1}$. Z předchozího příkladu víme, že $\mathcal{L}_{SG1} \subset \mathcal{L}_1$, nicméně to není příliš užitečná informace, jelikož víme, že jakýkoliv jazyk definovaný dle stavové gramatiky s omezenou hloubkou, tedy *n-limited* SG patří do nekonečné hierarchie tříd jazyků mezi třídami $\mathcal{L}_2$ a $\mathcal{L}_1$, tvořené postupným přidáváním jednotlivých *n-limited-SG*. A jelikož žádný počítač nemá neomezenou operační paměť, nikdy síly třídy $\mathcal{L}_1$ nedosáhne. Tedy víme, že $\mathcal{L}_{n-limited-SG1} \subset \mathcal{L}_{n-limited-SG}$, což celkově znamená:

$$\mathcal{L}_2 \subset \mathcal{L}_{n-limited-SG1} \subset \mathcal{L}_{n-limited-SG} \subset \mathcal{L}_{\infty-limited-SG} = \mathcal{L}_1. \tag{4.5.1}$$

Je zřejmé, že toto omezení je nežádoucí a tvoří velice silný argument proč zahrnout ε-pravidla. Na druhou stranu, ε-pravidla nemusí být nutně jediným možným řešením. Jeden takový případ bude rozebrán v kapitole 8, protože v této práci na jeho prozkoumání již nezbylo místo.

Kapitola 5

Hluboká syntaktická analýza

V této kapitole budou formálně definovány pojmy stavové LL-gramatiky a hierarchické syntaktické analýzy. Základní kostra těchto pojmů vzejde z definic obsažených v kapitole 2, které budou doplněny o vlastnosti popsané napříč touto prací.

Definice 5.0.1. Stavová LL-gramatika G je osmice (Q, Σ, Γ, P, S, q_0, n, l), kde:

- Q je konečná množina stavů.
- Σ je konečná množina vstupní abecedy (terminálů).
- Γ je úplná abeceda, kde $\Gamma = \Sigma \cup N$, kde N je konečná množina neterminálů.
- $P \subseteq (Q \times N) \times (Q \times \Gamma^+)$ je konečná relace, přičemž místo $(q, A, p, B) \in P$ budeme psát $qA \to pB \in P$.
- S je počáteční symbol na zásobníku, tedy $S \in N$.
- q_0 je počáteční stav, tedy $q_0 \in Q$. (Doplněno z důvodu usnadnění implementace).
- n je limit stavové gramatiky G, tak jak byl popsán v definici 2.1.5.
- l je limit určující počet možných opakování jednoho pravidla v jednom stavu v algoritmech založených na prohledávání stavového prostoru.
- Navíc stavová LL-gramatika musí mít tyto vlastnosti:
  a) $\forall a \in \Sigma$ a $\forall A \in N$ existuje maximálně jedno pravidlo tvaru $dpA \to qB$, takové že $a \in First_d(B)$, kde $0 < d \leq n, B \in \Gamma^*$. Podmínka omezující faktorizační konflikty.
  b) $\forall r_1 : d_1p_1A_1 \to q_1B_1, r_2 : d_2p_2A_2 \to q_2B_2 : p_1 = p_2 \Longrightarrow d_1 = d_2$. V jednom stavu existují pouze pravidla operující nad jedinou hloubkou zásobníku (popsáno v podkapitole 4.5).
  c) V gramatice G neexistuje pravidlo, které je dostupné až po zopakování jakéhokoliv pravidla v jednom stavu více než l-krát (popsáno v podkapitole 4.2).

Implementace této práce navíc uvažovala podmínku jediného startovního pravidla, tedy jakákoliv reálně testovaná gramatika splňovala $|npA \to qB| = 1$, kde $A = S$ a S je startovní neterminál.

Kapitola 6

Implementace

Tato kapitola se zaměřuje na specifika implementace syntaktické analýzy vyplývající z použití hlubokých zásobníkových automatů, respektive z podpory kontextové syntaktické analýzy. Nicméně, nejprve zde bude vysvětlen pojem hierarchické hluboké syntaktické analýzy a co všechno tento systém přináší.

6.1 Hierarchická hluboká SA

Hierarchická hluboká syntaktická analýza formálně definovaná v definici 6.1.1 (dále jen hierarchická SA) slouží k zjednodušení návrhu stavových gramatik pro hlubokou analýzu, přičemž hierarchická SA přináší hned několik výhod oproti přímočarému rozšíření standardní LL-analýzy.

Definice 6.1.1. Hierarchická syntaktická analýza je pětice H = (I, G = $\{g_1, g_2, ..., g_n\}$, P = $\{p_1, p_2, ..., p_n\}$, g_0, p_0), kde:

- I je společný vstup nebo-li společný lexikální analyzátor pro všechny dílčí hluboké zásobníkové automaty.
- G je konečná množina stavových LL-gramatik tak jak je popisuje definice 5.0.1.
- P je konečná množina hlubokých zásobníkových automatů, kde $|G| = |P|$ a kde p_1 je definováno dle g_1, p_2 je definováno dle g_2 až p_n je definováno dle g_n.
- g_0 je stavová LL-gramatika definující počáteční hluboký ZA p_0, kde $g_0 \in G, p_0 \in P$.

Na moment ignorujme hierarchickou SA a shrňme co všechno je potřeba realizovat pro hlubokou LL-analýzu bez ε-pravidel:

a) Změna struktury LL tabulky, respektive klíče řádku.

b) Výpočet hloubek pravidel realizované stavové gramatiky.

c) Modifikace algoritmů *First* aby reflektovala předchozí body.

d) Realizuji-li zotavení z chyb, modifikace algoritmu *Follow*.

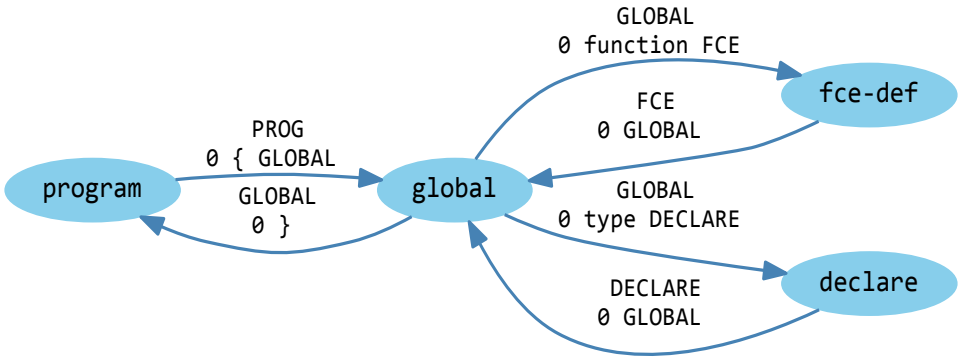
Body b, c a d evidentně vycházejí z bodu a, nicméně je důležité si uvědomit, že tyto tři body v sobě skrývají algoritmy, které operují nad stavovým prostorem. A právě zde má hierarchická SA největší přínos. V klasickém případě bychom navrhli nějakou stavovou gramatiku, která by byla v případě kompletního popisu běžného programovacího jazyka značně rozsáhlá a s tím i stavový prostor se kterým ve zmíněných algoritmech pracuje.

Takovou stavovou gramatiku je zřejmě možné definovat pomocí souboru malých stavových gramatik, mezi kterými by se v průběhu syntaktické analýzy dynamicky přecházelo. Nicméně to není všechno, jelikož bychom v takovém případě pracovali se souborem stavových gramatik, musí to znamenat, že budeme pracovat i se souborem hlubokých zásobníkových automatů, což sebou přináší zajímavé možnosti.

Hierarchická syntaktická analýza je tedy systém pracující nad souborem hlubokých ZA, které mají společný vstup (soubor s analyzovaným textem) a tedy i společný lexikální analyzátor, který žádají o *tokeny*. A podobně jako ZA, ať už klasický či hluboký, má hierarchická SA vstupní bod, což je v našem případě právě jeden z dostupných hlubokých ZA.

To má evidentně zřetelný přínos pro algoritmy zmíněné v bodě a-c, jelikož velikost stavové gramatiky se znatelně zmenší, respektive přesněji řečeno, její prohledávaný stavový prostor se zmenší, což nám umožňuje větší flexibilitu a volnost při návrhu stavových gramatik, zejména v souvislosti s problematikou jenž byla popsána v podkapitole 4.2.

Další přínos souvisí se zotavením z chyb. Z kapitoly 3 víme, že během klasické LL-analýzy je běžné v gramatice používat pravidla jako FOR → (DEC, EXPR, EXPR) { STATS } a DEC → type id ;, kde se koncové terminály '}' a ';', vygenerovaly hned na začátku analýzy dané syntaktické konstrukce. Díky tomu je také bylo možné použít jako záchytné body při výskytu chyby. Nastala chyba? Pokud ano, smaž vstup a zásobník až po koncový terminál a pokračuj. V našem případě to není možné, jelikož tyto neterminály se musejí přednostně rezervovat pro návrat ze syntaktické konstrukce a to díky absenci ε-pravidel.

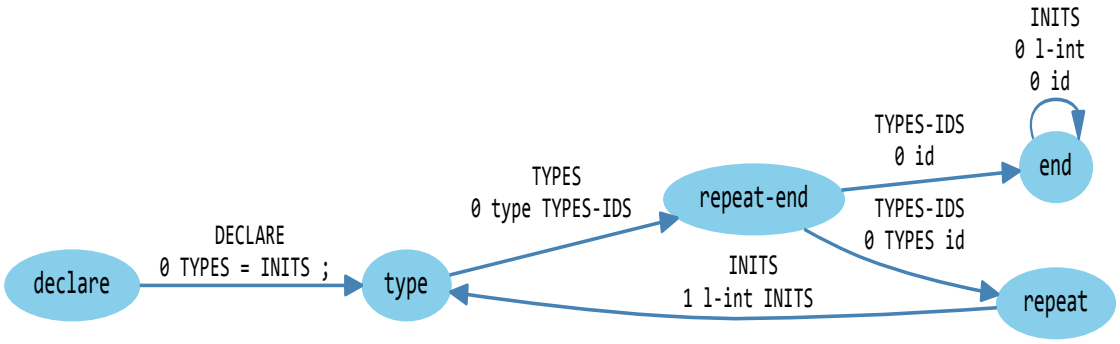


Obrázek 6.1: Stavová gramatika pro hlavní obálku programu.

Jak tedy hierarchická analýza může tento problém usnadnit? Představme si, že zrovna analyzujeme tělo funkce, konkrétně pak například deklaraci proměnné. V takovém případě předáme řízení syntaktické analýzy hlubokému ZA popisující deklaraci proměnné, který má k dispozici svůj vlastní zásobník. V případě chyby můžeme jednoduše celý tento zásobník zahodit, což znamená, že žádný záchytný bod na zásobníku není potřeba. Konkretně se

toto realizuje tak, že nezahodím jen zásobník, ale celý hluboký ZA, protože by stejně bylo nutné jeho stav resetovat. Samozřejmě to taky znamená, že při předání řízení nějakému HZA, se musí nejprve vytvořit kopie (klon) z databáze definovaných hlubokých ZA, jenž byly vytvořeny na základě vstupního souboru stavových gramatik.

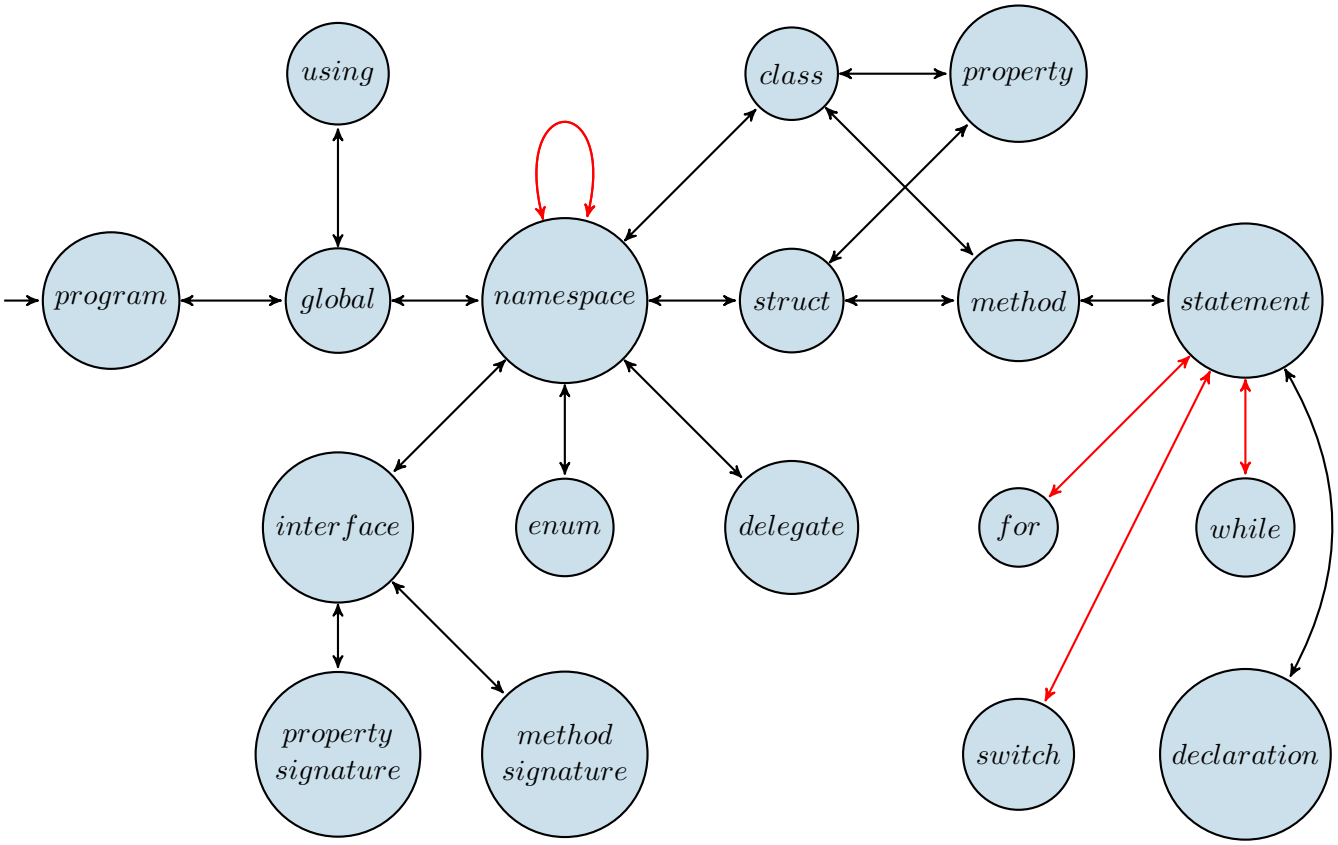
Pro předaní řízení z jednoho HZA do druhého a pro zachycení jejich hierarchie je zapotřebí zásobník, přičemž je už otázkou konkrétní implementace jak se takový zásobník realizuje. V této práci bylo zvoleno předání řízení pomocí rekurzivního sestupu, který zásobníkovou strukturu vytvoří automaticky v paměti počítače. To si samozřejmě vyžaduje, aby v definici stavové gramatiky bylo možné vyjádřit předání řízení. To bylo realizováno poměrně primitivním způsobem, kdy v případě, že se název stavu shoduje s názvem vstupního souboru gramatiky, jedná se o takzvaný předávací stav, vytvoří se kopie hlubokého ZA definovaného podle této gramatiky a zavolá se jeho metoda *Run*(). Podíváme-li se na obrázek 6.1, stav *fce-def* a stav *declare* jsou stavy pro předání řízení, konkrétní realizace jejich gramatik je vyobrazena na obrázcích 4.4 a 6.2.



Obrázek 6.2: Stavová gramatika definující kontextovou deklaraci proměnných.

Na obrázku 6.3 je znázorněno jak by mohla vypadat navigace v hierarchické syntaktické analýze. Daný příklad je inspirován jazyky z OO prostředí, konkrétněji pak jazykem C♯. Jednotlivé stavy zde představují samostatné hluboké ZA, definované na základně separátních definic stavových gramatik. Černé obousměrné šipky představují jaké syntaktické konstrukce daný hluboký ZA může obsahovat, přičemž jsou oboustranné, protože jakmile se dokončí syntaktická analýza, musí se řízení nutně předat zpět hierarchicky vyššímu HZA. Červené šipky pak naznačují, že daný jazyk konkrétního hlubokého ZA může obsahovat konstrukci ze které mu bylo předáno řízení. Tedy, nejen že muže vrátit řízení výše, ale také může předat řízení níže. Tato struktura se neimplementuje, jakékoli provázání je definováno v jednotlivých stavových gramatikách, jak je znázorněno na obrázcích 6.1 a 6.2. Samozřejmě, je zřejmé, že obrázek 6.3 není zdaleka kompletní.

Nakonec bych zmínil poměrně zřejmé vlastnosti, jako je značně zvýšená přehlednost definice stavových gramatik určených pro popis jednotlivých syntaktických konstrukcí jenž daný programovací jazyk oplývá. Velice užitečným nástrojem je i možnost přidat do vstupního souboru se seznamem gramatik libovolnou novou položku i když nepatří do daného celku a žádná z jiných gramatik ji nezahrnuje. Takto je možné ji velice snadno zahrnout do automatizovaného procesu kontroly gramatik tvořících hierarchickou SA.



Obrázek 6.3: Pohled na navigaci hierarchické hluboké syntaktické analýzy.

6.2 Zpracování vstupu

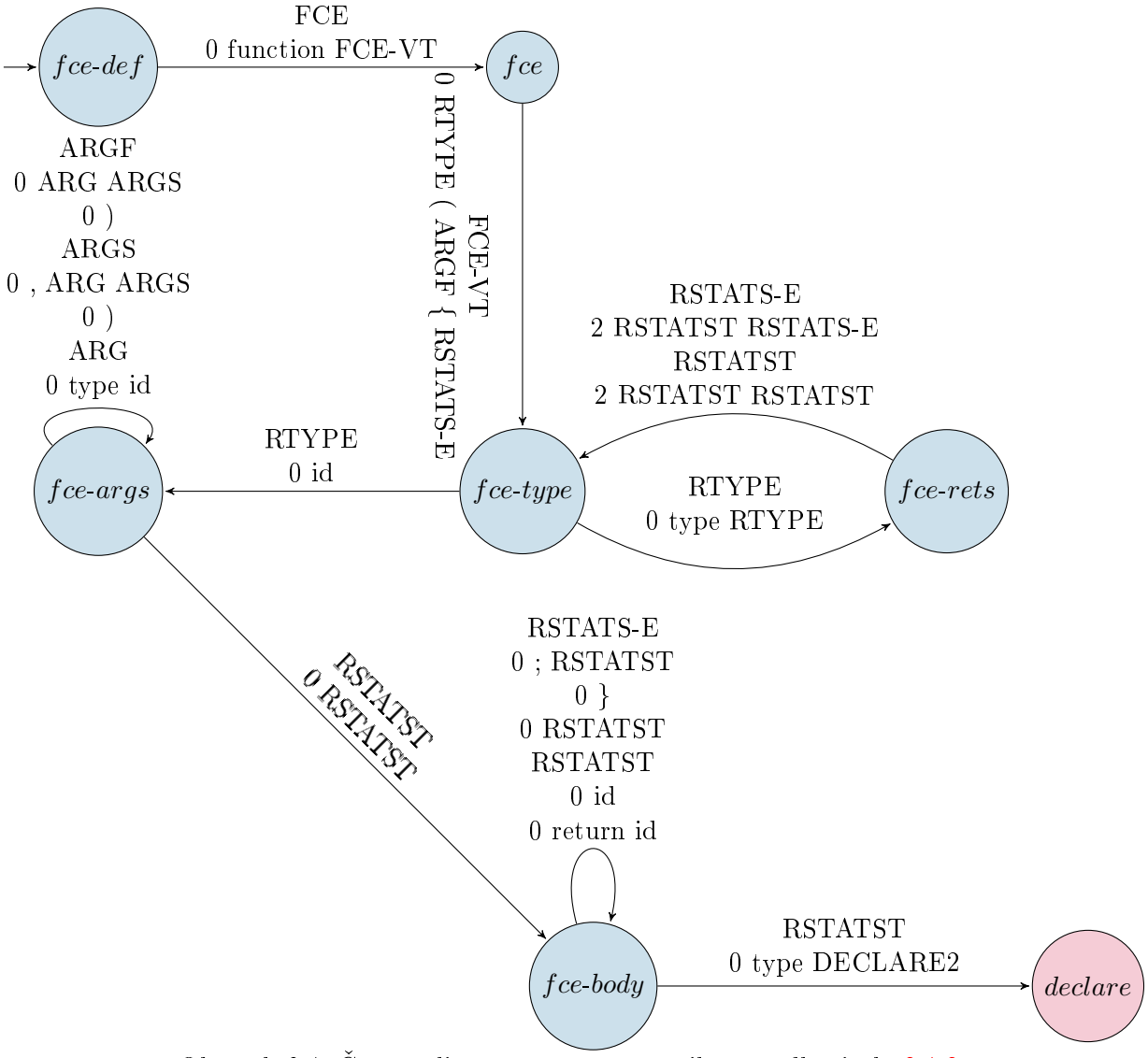
Při hluboké syntaktické analýze se vstup zpracovává velice podobně jako při běžné SA. Stále zde platí princip syntaxí řízeného překladu, kdy syntaktická analýza žádá lexikální analyzátor o další *tokeny*, kdykoli nějaký potřebuje. Nicméně u hluboké SA, na rozdíl od té klasické, je nutné přednačítání vstupu.

Přednačítání vstupu u překladačů není nic nového. Je běžné, že při jednom požadavku na *token* jich lexikální analyzátor načte vícero, ale vrátí pouze jeden, přičemž další *tokeny* načte jen tehdy pokud mu došly ty již přednačtené. Zcela jistě existuje více variací tohoto chování s více či méně odlišnostmi, nicméně důležitým faktem je, že toto chování je spíše volitelné rozšíření, implementované z důvodů efektivity a rychlosti překladu. Na druhou stranu u hluboké syntaktické analýzy je toto chování naprostou nutností.

Uvažujme například úryvek ze stavové gramatiky (obrázek 6.4) pro definici funkce zjednodušeně popsané v podkapitole 3.1.2. Pravidlo pro přechod ze stavu *fce* do stavu *fce-type* expanduje neterminál FCE-VT na řetězec (část větné formy) RTYPE (ARGF { RSTATS-E. Zároveň je neterminál RSTATS-E zpracovatelný ve stavu *fce-rets* pouze v hloubce dvě, což znamená, že abychom mohli toto pravidlo použít, musí se nejdříve správně namapovat část vstupu na neterminál ARGF. To znázorňuje obrázek 6.6.

Obecně mohou nastat dva případy. První je pokud jsou na zásobníku hlubokého ZA

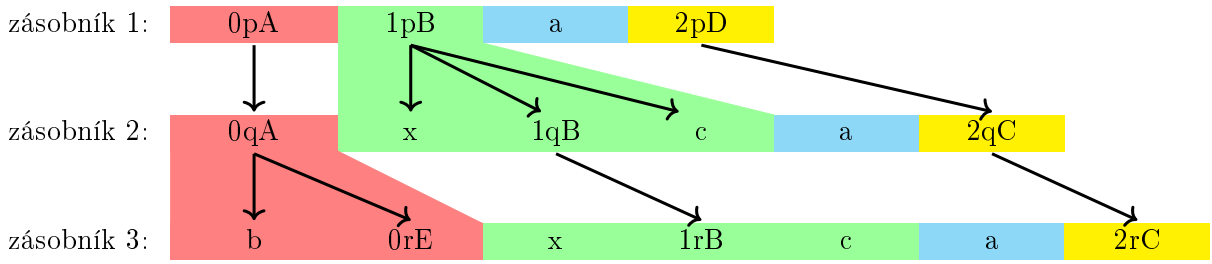
přítomny terminály, které ještě nebylo možné odstranit pomocí operace *compare*. V tom případě jednoduše kontroluji pokud jsou tyto neterminály postupně na vstupu. V opačném případě nastává chyba.



Obrázek 6.4: Část realizace stavové gramatiky z podkapitoly 3.1.2.

Druhým případem je výskyt neterminálu, právě jako například neterminál ARGF z obrázku 6.6, který se zpracovává až po neterminálu RSTATS-E. Zde nutně vyvstává otázka: Na jaké všechny terminály se může ARGF expandovat a ještě lépe, v jakém pořadí. Nicméně to je v podstatě samotná syntaktické analýza, proto tuto část kontroly provedeme jen zjednodušeným způsobem. Definujeme si množiny $Expansion_d(npA) = \{a \in W | npA \Rightarrow^+ qw$, kde $w \in \Sigma^*$ a kde W je množina všech terminálů ve větě $w\}$. Jinak řečeno, definujeme si množinu podobnou například $First_d$ či $Follow_d$, která bude určovat množinu všech terminálů, kterých trojice npA může nabýt. Výpočet množin $Expansion_d$ je částečně podobný výpočtu množin $Follow_d$, hlavním rozdílem je, že místo dvojic (předchůdce, následník) v

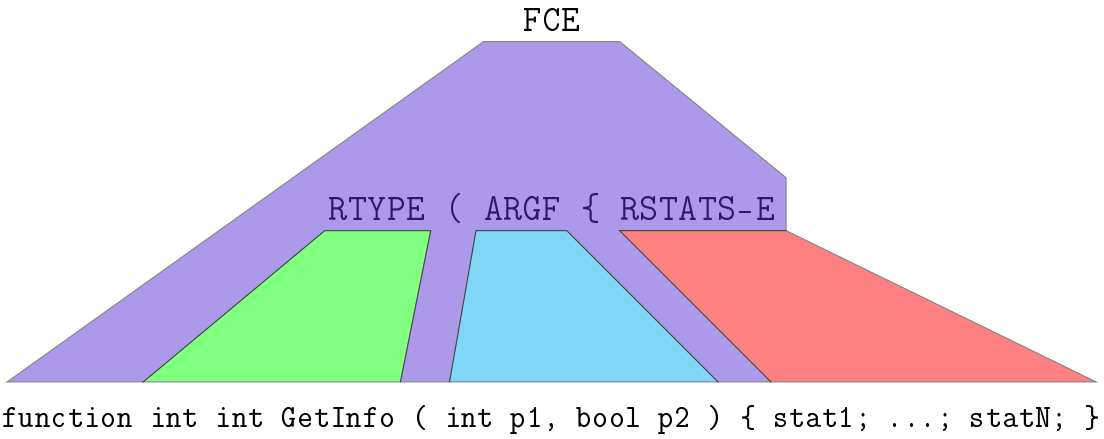
jedné konfiguraci zásobníku hlubokého ZA, se do seznamu L vkládají dvojice (předchůdce, následník) mezi všemi po sobě následujícími konfiguracemi HZA. Obrázek 6.5 znázorňuje princip výpočtu, přičemž černé šipky znázorňují páry přidávané do seznamu L, ze kterého jsou následným pospojováním těchto dvojic získány jednotlivé množiny $Expansion_d$.



Obrázek 6.5: Ilustrace výpočtu množin $Expansion_d$.

Vraťme se k obrázku 6.6, ze kterého je zřejmé, že symbol začátku těla funkce '{' musí být úspěšně namapován již před expandováním RSTATS-E, jinak by mohlo dojít k chybě. Například v případě, kdy úsek vstupu náležící ARGF by byl částečně totožný s RSTATS-E, by mohla nastat situace taková, že RSTATS-E by bylo expandováno pro neplatnou část vstupu, což by velice rychle vedlo na chybnou indikaci syntaktické chyby.

Pokud tedy mapujeme řetězec RTYPE (ARGF RSTAT-E na zásobníku vůči vstupu, protože potřebujeme provést expanzi neterminálu RSTATS-E, tak pokud všechny symboly mezi (a {, tedy pokud znaky náležící neterminálu ARGF nepatří do množiny $Expansion_d$ ($npARGF$), nastane chyba typu neočekávaný *token*, která bude poznamenána do chybového logu a bude zahájeno zotavení z chyby.



Obrázek 6.6: Znázornění mapování neterminálů na vstupní řetězec.

6.3 Zotavení z chyb založené na HSA

Jak už bylo řečeno výše, zotavení z chyb je jen vedlejším úkolem této práce, proto byl použit princip jednodušší než ten popsaný v podkapitole 4.5, a který byl založený na hledání symbolů náležících do množiny $Follow_d$, popřípadě $First_d \cup Follow_d$.

Pro jednoduchost byl použit triviální systém zotavení, kdy se ve vstupním řetězci vyhledává předem dohodnutý znak nebo skupina znaků. V této práci byly konkrétně použity symboly `';'` a `'}'`, jelikož ve všech testovacích gramatikách byl použit programovací jazyk velice podobný C či C++.

Uvažme vstupní řetězec `int bool a b = 5 true;` jehož syntaxi popisuje gramatika na obrázku 6.2, a který je bezchybný. V Řetězci `int bool a b 5 true;` naopak chybí operátor přiřazení, který je vygenerován pomocí pravidla `0<declare>DECLARE` → `TYPES = INITS ;`. Je zřejmé, že chyba nastane během provádění *compare* jelikož na zásobníku máme operátor přiřazení `'='` a zároveň na vstupu dostaneme číselný literál. V takovém případě se do chybového logu poznamená chyba typu neočekávaný *token*, vyhledá se nejbližší symbol `';'` nebo `'}'` a z tohoto místa se syntaktická analýza restartuje.

Sama o sobě tato metoda není příliš přesná. Například v řetězci `int bool a b = 5 true` chybí symbol ukončení příkazu `';'` což samozřejmě povede na přeskočení alespoň jednoho dalšího příkazu (a to i například celé smyčky). Zároveň hrozí přeskočení *tokenu* startu nového bloku (kódu) `'{'`, což může vést na chybné zotavení, jelikož se jednoduše můžeme ocitnout v hierarchii oborů platnosti o dvě úrovně níže. Tento problém byl vyřešen pomocí jednoduché inkrementace a dekrementace při výskytu znaků `'{'` a `'}'`.

K tomu navíc přibývá i jeden nový druh chyby, který u klasického zotavení z chyb nepotkáme. V podkapitole 6.2 byla popsána problematika přednačítání vstupu, která také může vést na chybu typu neočekávaný *token* mimo vrchol zásobníku, tedy v hloubce větší jedné (větší nule, pokud indexujeme pravidla od nuly).

Během zotavení z chyb se musí provést dva základní kroky. Prvním je vymazání chybné části vstupního řetězce až po nalezený záchytný symbol (';' a '}'), který je poměrně přímočarý. Na druhou stranu vymazání odpovídající části větné formy ze zásobníku je už těžší, především protože ne vždy máme onen záchytný symbol v daný moment již na zásobníku obsažený, což může být například způsobeno vyloučením ε-pravidel, tak jak to popisuje podkapitola 3.1. Řešení je použití hierarchické SA a zahození právě aktivního hlubokého ZA, tak jak to bylo popsáno podkapitole. 6.1.

6.4 Detaily implementace

V podkapitole 6.1 byl vysvětlen pojem hierarchické (hluboké) syntaktické analýzy jenž principiálně pracuje ve dvou hlavních bodech:

- Inicializace hierarchické syntaktické analýzy.
  a) Načtení všech gramatik $g_1, g_2, ..., g_n$ a vytvoření jim odpovídajících hlubokých zásobníkových automatů $m_1, m_2, ..., m_n$, kde n je počet stavových gramatik definujících HSA.
  b) Zavolání metody `Run()` počátečního HZA.
- Běh hlubokého zásobníkového automatu (algoritmus 6.4.1, metoda `Run()`).

Algoritmus 6.4.1. Princip metody `Run()` (pseudokód).

```
01: while (stack.Count > 0) {
02:   var key = GetLLKey();
03:   if (key == null)
04:     ErrorRecovery();
05:   Expand(key);
06:   if (currentState != startState && IsTransitState(currentState)) {
07:     var newPda = HSA.GetPda(currentState);
08:     newPda.Run();
09:   }
10:   Compare();
11: }
```

V prvním bodě je velice stručně popsána část práce hierarchické SA. Ve skutečnosti však tyto dva body jsou velice blízko reálné implementaci, protože HSA nemá za úkol nic jiného, než iniciovat syntaktickou analýzu. Kromě toho ještě nepatrně pomáhá s předáním řízení hlubokému ZA níže v hierarchii HZA.

V druhém bodě je k dispozici značně zidealizovaná podoba funkce `Run()`, přesto však zachycuje její podstatu. Idea metody je poměrně jednoduchá: dokud není zásobník právě aktivního HZA prázdný, získá se klíč do hluboké LL-tabulky a provede se expanze na základě pravidla jenž odpovídá danému klíči s následným provedením porovnání terminálních znaků na zásobníku a vstupu. V případě, že aktuální stav po expanzi je tranzitním stavem pro jiný hluboký ZA, získám jeho novou kopii a rekurzivně zavolám opět metodu `Run()`.

Vraťme se nyní k řádku číslo tři, kde voláme metodu `GetLLKey()` za níž se také ukrývá značná část logiky hluboké syntaktické analýzy. Tu zobrazuje algoritmus 6.4.2.

Algoritmus 6.4.2. Princip metody `GetLLKey()` (pseudokód).

```
01: var depth = stateDepths[currentState];
02: var nt = GetIthNonterminal(depth);
03: if (nt == null)
04:   threw new Exception(...);
05: var possibleTerms = <all terminals in deep-LL-table associated with
                         left side equal to (depth, currentState, nt)>;
06: if (possibleTerms.Count == 0)
07:   threw new Exception(...);
08: var inputTerm = <obtain the input terminal behind nonterminal nt>;
09: if (!possibleTerms.Contains(inputTerm)) {
10:   threw new Exception(...);
11: }
12: LLKey = <obtain LLKey associated with terminal inputTerm>;
13: return var LLKey;
```

Algoritmus funguje tak, že nejprve získá hloubku, což je možné a velice jednoduché díky omezení jediné hloubky pravidel operujících z téhož stavu. Následně na základě získané hloubky získá neterminál ze zásobníku aktuálně aktivního HZA. Pro kombinaci `(hloubka, aktuální-stav, neterminál)` získá všechny možné terminální symboly, jenž jsou obsažené v klíčích hluboké LL-tabulky, a které jsou spárovány právě s touto trojicí. Pokud žádné takové symboly neexistují, neexistuje ani použitelné pravidlo, což může být způsobeno buď chybou vstupu, načež se zahájí zotavení z chyb, nebo chybou v gramatice.

V opačném případě získáme vstupní terminál, který se ale ve vztahu k zásobníku nachází až za neterminálem `nt` a pokud se shoduje s nějakým ze symbolů v `possibleTerms`, bylo nalezeno pravidlo, které se také vrátí jako výsledek funkce.

Pozastavme se ještě u řádku číslo osm, za kterým se schovává kód (funkce), ve kterém dochází k přednačítaní vstupu popsaném v podkapitole 6.2. V praxi se na řádku osm vyskytuje funkce přebírající index do společného, přednačteného vstupu, což zabraňuje vyhledání nekorektního pravidla.

6.5 Použití aplikace

Výsledkem této práce je konzolová aplikace `deep-sa.exe`, která přijímá tři až pět parametrů. Následují tyto možnosti:

```
deep-sa.exe [--debug=N][--breadthSearchLimit=M] tokens hfile input
```

První parametr `--debug=N` je volitelný a může nabývat hodnot $N = 0$ až $N = 2$, kde 0 zamezuje výpisu ladících informací (výchozí nastavení), při 1 budou vypisovány základní ladící informace a při 2 pokročilé ladící informace.

Druhý volitelný parametr `--breadthSearchLimit=M` je nastavení limitu pro algoritmy prohledávající stavový prostor, tedy se jedná především o algoritmy pro výpočet hloubek pravidel a množin $First_d$ a $Follow_d$. Výchozí nastavení je $M = 2$, což je také minimální možná hodnota.

Další tři parametry jsou povinné, musí být uvedeny až za volitelnými parametry a musí být použity v uvedeném pořadí. Všechny tři představují vstupní soubory, kde:

`token` - Vstupní soubor definující podobu tokenů pro lexikální analyzátor.

`hfile` - Vstupní soubor se seznamem vstupních gramatik pro hierarchickou hlubokou syntaktickou analýzu.

`input` - Vstupní soubor, který chceme analyzovat.

Příklad 1 demonstruje formát vstupního souboru `tokens`. Každá položka zabírá dva řádky, kde v prvním řádku část před dvojtečkou je název tokenu. Část za dvojtečkou může nabývat hodnot 1 a 2, kde 1 udává, že se jedná o výčtový či intervalový typ a 2 že se jedná o konstantní symbol, či řetězec. Druhý řádek je regulární výraz definující podobu tokenu. Jediné na co je třeba dát si pozor je posloupnost tokenů se stejným prefixem. Delší by měly předcházet ty kratší, zrovna tak jak je to vidět v uvedeném příkladě při definici tokenů pro operátory umocnění a násobení (řádky 9-12).

```
01: l-double:2
02: ([-]?)(([0-9]+)(e[0-9]+)|([0-9]*[.][0-9]+)(e[0-9]+)?)
03: l-int:2
04: ([-]?)([0-9])+
05: l-bool:2
06: (true|false)
07: plus:1
08: [+]
09: power:1
10: [*]{2}
11: times:1
```

```
12: [*]
```

Příklad 1: Příklad vstupního souboru, který definuje tokeny.

Příklad 2 znázorňuje formát souboru `hfile` se seznamem gramatik pro hierarchickou SA. Každá položka je definována na jednom řádku, kde část před dvojtečkou je název gramatiky a část za dvojtečkou je název souboru tuto gramatiku definující. Položka s názvem start se použije jako definice pro hlavní (startovní) hluboký zásobníkový automat.

```
01: start:program.grammar
02: declare:declare.grammar
03: fce-def:fce-def.grammar
04: test:test.grammar
```

Příklad 2: Příklad vstupního souboru se seznamem gramatik pro hierarchickou SA.

Příklad 3 je ukázkou formátu souboru pro definici gramatiky (například `program.grammar`). Na začátku souboru (ne nutně) jsou definovány základní vlastností gramatiky, kde řádek uvozený `terms` je seznam terminálních symbolů definovaných v souboru `tokens` a oddělený čárkami (bez mezer), `nonterms` je seznam neterminálů (uppercase), `start-nonterm` určuje počáteční neterminál gramatiky, `states` je seznam jmen stavů, `start-state`, počáteční stav a `limit` je limit hloubky pravidel v gramatice. Například při $limit = 2$ jsou možné hloubky 1 a 2.

Dále následuje seznam samotných pravidel ve tvaru `<stav><neterminál> -> <stav><řetězec znaků zásobníkové abecedy>`. Navíc, řádek začínající znakem '#' je považován za komentář a neuvažuje se.

```
01: terms:type,id,l-int,=,;
02: nonterms:DECLARE,TYPES-IDS,TYPES, INITS
03: start-nonterm:DECLARE
04: states:declare,type,end,repeat-end,repeat
05: start-state:declare
06: limit:2
07: # samotna pravidla
08: <declare>DECLARE -><type>TYPES = INITS ;
09: <type>TYPES -><repeat-end>type TYPES-IDS
10: <repeat-end>TYPES-IDS -><repeat>TYPES id
11: <repeat-end>TYPES-IDS -><end>id
12: <repeat>INITS -><type>l-int INITS
13: <repeat>INITS -><type>id INITS
14: <end>INITS -><end>l-int
15: <end>INITS -><end>id
```

Příklad 3: Příklad vstupního souboru s definicí gramatiky pro deklaraci proměnné.

Konečně, soubor `input` je vstupním souborem, který chceme analyzovat. Například pokud by startovní gramatikou byla gramatika uvedená v příkladě 3, mohl by obsahovat řetězec:

```
int bool int num1 flag num2 = 5 variable 1020;
```

Kapitola 7

Syntaktické konstrukce hluboké SA

Tato kapitola představí několik gramatik, které mohou přijímat kontextové syntaktické konstrukce, přičemž nejprve bude u každého případu uveden příklad vstupních řetězců, pak stavová gramatika definující jazyk do nějž uvedené řetězce patří. Dále bude uveden hluboký zásobníkový automat definovaný dle uvedené gramatiky a nakonec, pokud to bude vhodné, bude uveden příklad práce daného HZA nebo krátké vysvětlení jeho principu.

7.1 Matice

Jako první příklad se zjevně nabízí gramatika umožňující přijímat řetězce typu

```
int string bool number str flag = 1234 "It's a nice day." true,
```

ale protože gramatika pro tento typ deklarace proměnných byla použita opakovaně napříč touto prací, uvažme například gramatiku, která by přijímala tyto vstupní řetězce:

Příklad vstupu

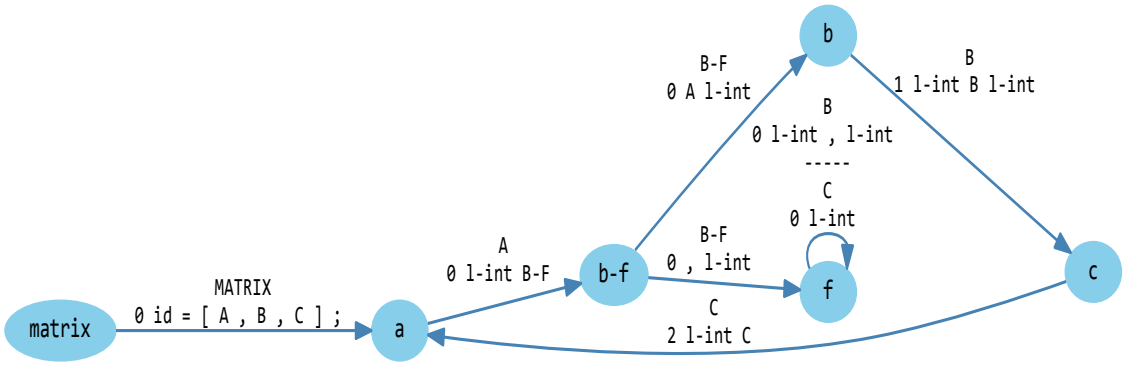
```
matrix = [
    1 0 0 0,
    0 1 0 0,
    0 0 1 0,
    0 0 0 1,
    1 1 1 1
];
```

```
matice = [
    1 2 3 1 0 0 1 5 6,
    5 6 7 8 5 8 7 4 9,
    9 1 1 0 4 5 7 8 8,
    1 8 9 7 7 8 5 8 9,
    1 5 6 7 7 8 1 1 0
];
```

Stavová gramatika

```
<matrix>MATRIX -> <a>id = [ A , B , C ] ;
<a>A -> <b-f>l-int B-F
<b-f>B-F -> <b>A l-int
<b-f>B-F -> <f>, l-int
<b>B -> <c>l-int B l-int
<f>B -> <f>l-int , l-int
<c>C -> <a>l-int C
<f>C -> <f>l-int
```

Hluboký zásobníkový automat



Popis

Gramatika pracuje na stejném principu jako ta definující výše zmíněný jazyk pro deklaraci proměnných, tedy kontroluje všechny řádky matice paralelně v tom smyslu, že nejdříve otestuje první token prvního řádku, pak první token druhého řádku, atd. Podstatné je, že každá matice náležící do přijímaného jazyka musí mít přesně pět řádku. Gramatiku pro matici s libovolnými rozměry $m \times n$ není kvůli omezení na jednu hloubku pravidel pro tentýž stav gramatiky možno sestrojit.

Kapitola 8

Další směry výzkumu

V této kapitole se zaměříme na další možné směry vývoje či výzkumu vyplývající z této práce. Zejména zde budou alespoň teoreticky navrhnuty algoritmy pro výpočet množin $Empty_d$, $First_d$ a $Follow_d$ pro hlubokou LL-syntaktickou analýzu zahrnující ε-pravidla, protože to se jeví jako neperspektivnější budoucnost hluboké LL-analýzy. Předtím zde však bude popsáno jedno možné řešení problému jediné hloubky pravidel na stav hluboké LL-gramatiky (HZA), které s ε-pravidly nemá nic společného.

8.1 Makro neterminální symboly

Již víme, že pokud stavová gramatika ztratí schopnost disponovat pravidly s libovolnými hloubkami pro jeden stav, ztrácí zároveň schopnost sekvenční kontroly relací mezi kontrolovanými částmi vstupního řetězce. Například jazyk $L_{complex} = \{(a^n,)^{k-1}(a^n); |n \geq 1 \wedge k \geq 2\}$ (takto není třeba ošetřovat okrajové podmínky) není možné analyzovat gramatikou pracující na paralelním principu, jelikož jazyk $L_{complex}$ může obsahovat libovolně mnoho sekvencí. Částečným řešením by bylo sestavit několik gramatik pro různé hloubky a k vhodně omezit. Jiným řešením by mohla být dynamická generace gramatiky až za běhu syntaktické analýzy. Konečně, možné řešení, které nás zajímá a které tato práce již nestihla prozkoumat je teorém 8.1.1.

Hypotéza 8.1.1. *Každá ničím nepřerušená sekvence totožných neterminálů bude zabírat jedinou hloubku h na zásobníku.*

Zapomeňme teď na moment na omezení jediné hloubky pravidel na stav a mějme gramatiku definující jazyk $L_{complex}$:

```
<s>S → <q>X E          <f>P → <f>a
<q>X → <b>a X P        <c>E → <d>A E
<b>X → <b>a X P        <d>P → <e>a
<b>X → <c>,            <e>A → <d>A P
<c>E → <f>;            <d>A → <c>,
```

Zároveň uvažme teoretické sekvence konfigurací HZA při analýze vstupních řetězců jazyka $L_{complex}$ (i) `"a,a,a;"`, (ii) `"aa,aa,aa;"` a (iii)`"aaa,aaa,aaa;"`, přičemž jednotlivé konfigurace budou zapsány ve formátu `<stav>:<zásobník>` a pro zlepšení názornosti vynecháme *compare* kroky.

ad (i)
s:S ⇒
q:XE ⇒
b:aXPE ⇒
c:a,PE ⇒
d:a,PAE

ad (ii)
s:S ⇒
q:XE ⇒
b:aXPE ⇒
b:aaXPPE ⇒
c:aa,PPE ⇒
d:aa,PPAE

ad (iii)
s:S ⇒
q:XE ⇒
b:aXPE ⇒
b:aaXPPE ⇒
b:aaaXPPPE ⇒
c:aaa,PPPE ⇒
d:aaa,PPPAE

Z posloupností (i),(ii) a (iii) je zřejmé, že kdykoliv syntaktická analýza dosáhne stavu c, provede se pravidlo <c>E → <d>A E a to pouze tehdy, když za sekvenci neterminálů P je další znak ',' a tedy se předpokládá, že musí následovat další řádek. Podstatné však je, že toto pravidlo se vždy provedlo v rozdílné hloubce, vlivem proměnné délky sekvence neterminálů P.

Vraťme se nyní k hypotéze 8.1.1. Uvážíme-li, že algoritmus pro získání hloubek pravidel (4.2.1) v nepřerušované sekvenci totožných neterminálů vždy z důvodů efektivity vybere první (při SA se snažíme co nejdříve dosáhnout terminálů na vrcholu zásobníku), můžeme v daný moment zbytek sekvence ignorovat, respektive pokud implementujeme datový typ neterminálu například takto:

```
01: class Nonterminal : AbstractStackSymbol
02: {
03:     string name;
04:     int length;
05:     ...
06: }
```

pak z pohledu algoritmu 4.2.1 neztratíme potřebné informace, a zároveň to zabrání variabilní hloubce pravidel (<c>E → <d>A E), která nutně musí pracovat s neterminálem až za sekvencí neterminálů s proměnnou délkou. Samozřejmě se tímto mimo jiné uspoří i operační paměť. Na druhou stranu je potřebná relativně rozsáhlá reimplementace hluboké SA.

Pokud tedy zopakujeme hlubokou SA z předchozího příkladu s použitím zápisu A(n) ($A \in N, n \geq 1$), který vyjadřuje, že délka nepřerušované sekvence neterminálů A je n (sekvence délky jedna zapíšeme jako samotný neterminál), pak je zřejmé, že pravidlo <c>E → <d>A E se vždy použije v hloubce dvě (nezapomeňme, že terminály se ignorují, respektive v tomto příkladě byly vynechány *compare* kroky).

ad (i)
s:S ⇒
q:XE ⇒
b:aXPE ⇒
c:a,PE ⇒
d:a,PAE

ad (ii)
s:S ⇒
q:XE ⇒
b:aXPE ⇒
b:aaXP(2)E ⇒
c:aa,P(2)E ⇒
d:aa,P(2)AE

ad (iii)
s:S ⇒
q:XE ⇒
b:aXPE ⇒
b:aaXP(2)E ⇒
b:aaaXP(3)E ⇒
c:aaa,P(3)E ⇒
d:aaa,P(3)AE

Dokončení (ii): ... ⇒ e:aa,aPAE ⇒ d:aa,aPAPE ⇒ e:aa,aaAPE ⇒ d:aa,aaAP(2)E ⇒ c:aa,aa,P(2)E ⇒ f:aa,aa,P(2); ⇒ f:aa,aa,aP; ⇒ f:aa,aa,aa;

8.2 Přidání ε-pravidel

Hlavním problémem při zahrnutí ε-pravidel do hluboké LL-analýzy a současně i důvod proč je tato práce neuvažovala, je skutečnost, že definice hlubokého zásobníkového automatu [6] je taktéž ignoruje, což odpovídá i definici stavové gramatiky [9].

Na druhou stranu, bylo prokázáno, že stavové gramatiky s ε-pravidly se svojí generativní silou rovnají turingovu stroji [7]. To samé se však dosud nepodařilo dokázat i pro hluboký zásobníkový automat, což tvoří jednu z hlavních oblastí pro budoucí výzkum.

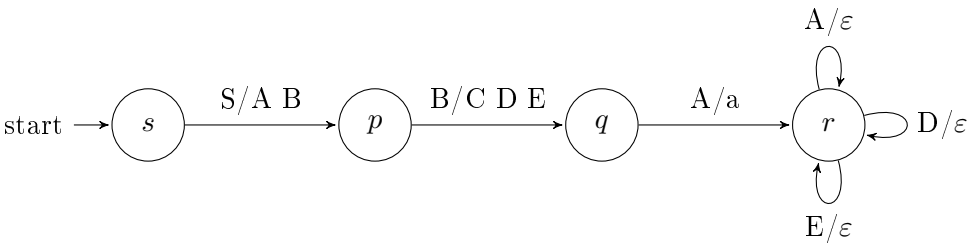
8.2.1 Množiny Empty_d, First_d, Follow_d

Vyjděme z algoritmu 8.2.1 pro výpočet množiny *Empty* u klasické LL-analýzy, který funguje na zpětném principu. Tedy, nejdříve se zjistí ty neterminály, které jsou smazatelné přímo pomocí jednoduchých pravidel a následně se z nich skládají ty složitější. Podobným způsobem by mohl fungovat i algoritmus pro výpočet množiny $Empty_d$, který opět bude pracovat na principu prohledávání stavového prostoru.

Algoritmus 8.2.1. Výpočet množin *Empty* u klasické LL-analýzy.
Vstup: Bezkontextová LL- gramatika G = (N, T, P, S).
Výstup: Množina $Empty(a)$ pro každé $a \in \Sigma \cup N$.

1. $\forall a \in \Sigma : Empty(a) = \varnothing$.
2. $\forall A \in N : A \to \varepsilon \in P \Rightarrow Empty(A) = \{\varepsilon\}$, jinak $Empty(A) = \varnothing$.
3. Pokud $A \to B_1 ... B_n (B_i \in \Sigma \cup N) \wedge Empty(B_i) = \{\varepsilon\}$ pro všechna $i \in \{1, ..., n\}$, pak $Empty(A) = \{\varepsilon\}$.
4. Používej pravidlo 3. dokud se některá množina *Empty* mění.

Množina $Empty_d$ pro terminální symboly není potřeba řešit. Pro neterminální symboly, respektive složené klíče uvažujme každé pravidlo stavové gramatiky $r \in P$ jako startovní a simulujme pro něj syntaktickou analýzu podobně jako v algoritmu 4.2.1. Pokud se zásobník podaří kompletně vymazat, zjistili jsme že levá stranu pravidla r, že je smazatelná. A to nám tedy umožní zjistit množiny $Empty_d$ pro všechny složené klíče, které jsou smazatelné přímo, nebo složením těchto jednoduchých pravidel. Nicméně toto není dostatečné, protože některé neterminály mohou být smazatelné jen pokud je k dispozici neterminál (třeba i nesmazatelný) z předchozích kroků SA (obrázek 8.1), k čemuž je nutná simulace SA za použití opravdového startovního pravidla.



Obrázek 8.1: Ilustrační stavová gramatika pro výpočet množiny $Empty_d$.

Na obrázku 8.1 je zřejmé, že ve stavu q musí být k dispozici neterminál A aby bylo možné ve stavu r smazat neterminály C, D a E a tedy určit neterminál B (1pB u HZA) jako

smazatelný. Celkově to tedy znamená, že pro výpočet množin $Empty_d$ je potřeba simulovat SA klasicky ze startovního pravidla a jakmile se použije nějaké pravidlo r s levou stranou, která ještě nebyla deklarována jako smazatelná, musí se od tohoto momentu sledovat úsek zásobníku, který byl expandován pravidlem r, zda-li nebyl smazán, respektive, jestli jeho délka následnými expanzemi nebyla redukována na nulu. Dá se čekat, že implementace tohoto algoritmu nebude příliš jednoduchá.

Algoritmy $First_d$ a $Follow_d$ by mělo pro přidání podpory ε-pravidel stačit rozšířit podobně jako u klasické LL-analýzy, tedy pro $First_d$ zajistit, že pokud je první neterminál v expanzi pravidla $r : pA \to qB_1B_2$ smazatelný, přidáme do $First_d(pA) = First_d(qB_1)$ i $First_d(B_2)$, atd. Velice podobně se musí modifikovat i algoritmus pro $Follow_d$.

Kapitola 9

Závěr

V rámci této práce byla navržena a implementována hluboká syntaktická LL-analýza v rozsahu potřebném pro analýzu jazyků spadajících do třídy $\mathcal{L}_{n-limited-SG1}$, jenž představuje třídu jazyků založených na stavových gramatikách (hlubokých ZA), které jsou složeny pouze z pravidel operujícími nad jedinou hloubkou zásobníku v rámci jednoho stavu. Třída $\mathcal{L}_{n-limited-SG1}$ spadá mezi třídy jazyků $\mathcal{L}_2$ a $\mathcal{L}_1$. Pokud zahrneme i třídy reprezentované nedeterministickými a deterministickými HZA, dostaneme vztah 9.0.1. Uvědomme si však, že třída $\mathcal{L}_{DHZA}$ v rámci této práci a vztahu 9.0.1 odpovídá síle HZA definovaného podle gramatiky generující jazyk náležící do třídy $\mathcal{L}_{n-limited-SG1}$, tedy je velice pravděpodobné, že existují deterministické hluboké ZA s výrazně větší generativní silou.

$$\begin{aligned} \mathcal{L}_2 \subset \mathcal{L}_{n-limited-SG1} = \mathcal{L}_{DHZA} \subset \\ \subset \mathcal{L}_{n-limited-SG} = \mathcal{L}_{NHZA} \subset \mathcal{L}_{\infty-limited-SG} = \mathcal{L}_1. \end{aligned} \tag{9.0.1}$$

K tomu aby byla hluboká LL-analýza realizovatelná, byla v této práci navržena řada modifikací klasické LL-analýzy. Prvním krokem byla změněna struktury LL-tabulky tak, že řádkovým indexem není nadále již pouze neterminální symbol, ale trojice (hloubka-pravidla, stav, neterminál), z čehož samozřejmě vyplývá řada dalších změn.

Pro výpočet hloubky pravidel byl navržen a implementován algoritmus 4.2.1, který simuluje hlubokou syntaktickou analýzu na principu slepého prohledávání stavového prostoru do šířky s omezením na počet opakování téhož pravidla v jednom stavu. Pokud bychom toto omezení (či nějaké jiné) nezavedli, algoritmus by nikdy neskončil, protože pracujeme s nekonečnými jazyky a tedy je možné generovat nekonečně mnoho vět do nich náležících.

Dále byly navrženy a implementovány algoritmy pro výpočet množin $First_d$ (4.3.5) a $Follow_d$ (4.4.1), reprezentující hlubokou obdobu množin $First$ a $Follow$ klasické LL-analýzy, přičemž uvážíme-li $First_d(a) = s$, pak dohromady se změnou struktury hluboké LL-tabulky vyplývá, že $a =$ (hloubka-pravidla, stav, neterminál).

Jelikož hluboká LL-analýza obecně (teoreticky) operuje v rámci libovolné hloubky zásobníku, je nutné umožnit přednačítání vstupu, během kterého je zapotřebí kontrolovat, že se správná část vstupu použije pro odpovídající neterminál na zásobníku. Pro tento účel byl navržen algoritmus pro výpočet množiny $Expansion_d(a)$ popsaný v sekci 6.2 a který reprezentuje množinu všech terminálů expandovaných z trojice a.

Dalším z důležitých výsledků této práce je nutnost omezení pravidel pracujících v rámci jednoho stavu na jedinou hloubku zásobníku hlubokého ZA, protože v opačném případě není jasné, které pravidlo se má použít a hluboká syntaktická analýza se tak stává nedeterministickou. Dohromady tedy získáme strukturu LL-tabulky vyobrazenou v tabulce 9.1 a

formální definici stavové LL-gramatiky obsaženou v kapitole 5.

	a_1	a_2	$\cdots$	a_m
(n_1, s_1, S_1)	$\alpha((n_1, s_1, S_1), a_1)$	$\alpha((n_1, s_1, S_1), a_2)$	$\cdots$	$\alpha((n_1, s_1, S_1), a_m)$
(n_1, s_1, S_2)	$\alpha((n_1, s_1, S_2), a_1)$	$\alpha((n_1, s_1, S_2), a_2)$	$\cdots$	$\alpha((n_1, s_1, S_2), a_m)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
(n_1, s_2, S_1)	$\alpha((n_1, s_2, S_1), a_1)$	$\alpha((n_1, s_2, S_1), a_2)$	$\cdots$	$\alpha((n_1, s_2, S_1), a_m)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
(n_i, s_j, S_k)	$\alpha((n_i, s_j, S_k), a_1)$	$\alpha((n_i, s_j, S_k), a_2)$	$\cdots$	$\alpha((n_i, s_j, S_k), a_m)$

Tabulka 9.1: Teoretická podoba LL tabulky pro HZA, kde $\forall n_i \in \mathbb{N}$, $\forall s_j \in Q$, $\forall S_k \in N$, i, j, k, m ≥ 1 a $\forall a_m \in \Sigma$.

Výše zmíněné omezení má však za následek poměrně výrazné snížení generativní síly přijímaných jazyků, a proto se syntaktická analýza v této konkrétní podobě nejeví jako příliš praktická, respektive, její přínos by byl naprosto minimální. Především, uvážíme-li, že jedním z hlavních přínosů kontextové analýzy je úplné či alespoň částečné nahrazení sémantické analýzy založené na technikách jako je tabulka symbolů, pro kterou třída jazyků $\mathcal{L}_{n-limited-SG1}$ není dostatečně obecná.

Jako přirozené řešení se nabízí zahrnout ε-pravidla do stavové gramatiky definující HZA. Na druhou stranu, jejich zahrnutí nemusí být jediným řešením, jak zvýšit obecnost třídy jazyků přijímaných pomocí hluboké LL-analýzy. Jako perspektivní se jeví hypotéza 8.1.1 v kapitole 8, kterou pokud by se podařilo ověřit, by měla za následek výrazné snížení variability hloubek pravidel a zároveň zvýšení síly hluboké LL-analýzy zpět směrem k třídě $\mathcal{L}_{n-limited-SG}$.

Nicméně, i tak by bylo přidání ε-pravidel velice užitečné. Nejen protože se jedná o přirozenou nástavbu (rozšíření), ale hlavně protože návrh stavových gramatik pro přijímání kontextových jazyků, které jsou implicitně značně komplexní, je obtížné, a ε-pravidla by právě tuto situaci mohla výrazně zlepšit.

Konečně, součástí této práce je i popis, formální definice a implementace systému hluboké LL-analýzy nazvaný *hierarchická syntaktická analýza*, která opouští klasickou strukturu definice gramatiky v rámci jednoho jediného souboru. Naopak, každá jednotlivá konstrukce je definována v samostatném souboru. Výše popsané algoritmy je pak možné použít samostatně pro jednotlivé stavové gramatiky a na jejich základě definovat sadu hlubokých zásobníkových automatů, jejichž kopie jsou využívány během samotného procesu syntaktické analýzy. Pokud se během aktuálně analyzované syntaktické konstrukce vyskytne chyba, je aktuální hluboký ZA zahozen a řízení je navráceno nadřazenému v hierarchii hlubokých ZA, reprezentované zásobníkovou strukturou.

Literatura

[1] Allen, E.; Chase, D.; Luchangco, V.; aj.: *The Fortress Language Specification*. Sun Microsystems, Inc, April 26, 2005.

[2] Eikhout, V.: *TEXby Topics, a TEXnitian's Reference*. Addison-Wesley UK, 2001, iSBN 0-201-56882-9.

[3] Češka, M.; Vojnar, T.; Smrčka, A.: *Teorretická informatika - Studijní opora*. -, 2013.

[4] Meduna, A.: *Automata and languages : theory and applications*. London : Springer, 2000, iSBN 1-85233-074-0;ISBN 978-1-85233-074-3.

[5] Meduna, A.: *Formal languages and computation : models and their applications*. Boca Raton : CRC Press, 2014, iSBN 978-1-4665-1345-7.

[6] Meduna, A.: Deep pushdown automata. *Acta Informatica [online]*, ročník 42, 2006: s. 541–552.

[7] Meduna, A.; Horváth, G.: On state grammars. *Acta Cybernetica*, ročník 8, 1988: s. 237–245.

[8] Rábová, Z.; Hanáček, P.; Peringer, P.; aj.: Užitečné rady pro psaní odborného textu [online]. `http://www.fit.vutbr.cz/info/statnice/psani_textu.html`, 2008-11-01 [cit. 2008-11-28].

[9] Takumi, K.: An hierarchy between context-free and context-sensitive languages. *Journal of Computer and System Sciences [online]*, ročník 4, 1970: s. 492–508.