Pokročilé techniky LL parsingu

Radim Kocman

2015

Obsah

Kapitola 1

Úvod

Při studiu technik syntaktické analýzy byla metoda LL parsingu pravděpodobně jednou z prvních, která vám byla představena. Tato skutečnost není žádným překvapením, jde o nejjednodušší deterministickou metodu syntaktické analýzy. Všechny části LL parsingu lze s trochou snahy připravit i ručně bez pomoci automatizujících nástrojů a třída LL gramatik je dostatečně rozsáhlá na to, aby pokryla většinu základních konstrukcí programovacích jazyků. Jedná se tedy o ideální volbu pro uvedení do dané problematiky.

U pokračujícího studia se pak pozornost studijních materiálů velmi rychle přesune k LR parsingu. Jedná se opět o pochopitelný krok, LR gramatiky jsou oproti LL gramatikám silnější a obsáhnou i složitější konstrukce programovacích jazyků. Na druhou stranu se ale značně ztíží výpočet rozkladové tabulky a i samotná práce s takovouto tabulkou vyžaduje sofistikovanější řízení. Problémy s výpočtem a velikostí rozkladové tabulky jsou dokonce tak závažné, že se v praxi plný LR parsing prakticky nepoužívá a využívají se jen jeho aproximace, jakou jsou LALR nebo SLR.

Jen málo se ale v běžných studijních materiálech dočtete o pokročilejších variantách LL parsingu, které dokáží zlepšit vlastnosti syntaktického analyzátoru anebo umožní zpracování některých složitějších konstrukcí programovacích jazyků. Složitost práce s výslednou rozkladovou tabulkou přitom může zůstat téměř stejná. Daň za zlepšení se výrazněji projeví jen složitější metodou pro vytvoření takovéto tabulky. V knihách jsou však tyto pokročilé metody většinou popsány jen velmi abstraktně a při implementaci je nutné si různé detaily za pochodu domýšlet. Rozhodl jsem se proto využít svých zkušeností z vývoje nástroje pro generování LL(k) rozkladových tabulek[1] a v následujícím textu podrobněji představím rozličné pokročilé metody LL parsingu a to i s ohledem na jejich implementaci.

Nejspíše již víte, že celý název LL(1) se často zkracuje na LL a že (1) v názvu znamená predikci podle jednoho prvního symbolu z nezpracované části vstupního řetězce. Méně známé už je ale to, že běžná metoda výpočtu

[1] `http://www.fit.vutbr.cz/~ikocman/llkptg/`

LL(1) rozkladové tabulky je pouze aproximací plné LL(1) metody a přesně je tedy označována jako strong-LL(1). Pro sjednocení terminologie je tak v kapitole 2 nejdříve znovu představen běžný LL(1) parsing. Následující kapitola 3 se poté zaměřuje na plný LL(1) parsing a jeho rozdílný způsob konstrukce rozkladové tabulky. Kapitola 4 představuje LL(k) parsing, který rozšiřuje predikci na k prvních symbolů z nezpracovaného části vstupního řetězce. V kapitole 5 je představena modifikace LL(k) parsingu, kterou lze řídit zásobníkovým automatem čtoucím vstup po jednom symbolu. Závěr v podání kapitoly 6 pak ještě jednou shrnuje poznatky předchozích kapitol a v krátkosti nastiňuje další známe modifikace LL metod.

Kapitola 2

Základy LL parsingu

Následující text předpokládá, že čtenář je znalý základů z oblasti formálních jazyků a překladačů. Pojmy jako gramatika, zásobníkový automat nebo derivační strom jsou zde proto již používány bez bližšího seznámení. Zájemci mohou nalézt základy formálních jazyků a překladačů např. v knize [5]. Výklad o metodách LL parsingu je pak založen na knihách [3, 1] a skriptech [2]. Kniha [3] obsahuje aktuální souhrn metod LL parsingu, kniha [1] blíže popisuje tvoření LL(k) rozkladových tabulek a skripta [2] sloužila převážně pro ustanovení české terminologie.

Principem syntaktické analýzy je provedení rekonstrukce derivačního stromu zadaného vstupního řetězce podle předem známé gramatiky. Lineárně reprezentovaná informace tak může znovu získat strukturu, se kterou je poté možné dále pracovat. Ne všechny vstupní řetězce nutně patří do zadané gramatiky, musí tedy existovat způsob odmítnutí vstupu a případně i možnost zotavení po chybě ve stupu. Gramatiky se pak liší v obtížnosti nalezení derivačního stromu pro zadaný vstupní řetězec. Existují gramatiky, pro které je algoritmické nalezení derivačního stromu vstupního řetězce neřešitelné, dále pak gramatiky, kde je nalezení derivačního stromu značně obtížné, a jen specifická část gramatik umožňuje jednoduché deterministické nalezení derivačního stromu v lineárním čase. Pro praktické využití nás přitom zajímají převážně pouze poslední z nich a i ty se ještě liší v tom, jak složité je pro ně deterministickou metodu nalézt. Metody syntaktické analýzy pak ještě rozdělujeme podle způsobu, jakým se snaží derivační strom rekonstruovat. Ve výsledku tedy může a i existuje opravdu velké množství rozličných metod s různým zaměřením. Nejznámější a nejpoužívanější jsou deterministické metody pracující shora dolů a zdola nahoru. Do první kategorie spadají ty metody, které začínají s rekonstrukcí derivační stromu od počátečního neterminálu a snaží se systematicky vygenerovat vstupní řetězec; sem patří všechny LL metody. Do druhé kategorie spadají ty metody, které začínají s rekonstrukcí od vstupního řetězce a snaží se ho systematicky zredukovat na počáteční neterminál; sem patří všechny varianty LR metod.

2.1 Běžný LL(1) parsing

Podle předchozích rozdělení jde o deterministickou metodu pracující shora dolů, která ke své predikci využívá jeden první symbol z nezpracované části vstupního řetězce. Metody pracující shora dolů mají celkem omezené množství informací, které mohou při rozhodování o dalšího postupu využít. V případě běžného LL(1) parsingu pracujeme pouze s nejlevějším neterminálem v aktuální větné formě (reálně pak i s terminály) a oním prvním symbolem z nezpracované části vstupu. Za pomocí těchto informací je potom nutné sestavit postup zpracování celého vstupu.

Průběh metody si můžeme abstraktně představit následujícím způsobem. Na vstupu dostaneme gramatiku a vstupní řetězec. Začneme s derivačním stromem obsahujícím počáteční neterminál a budeme pracovat s funkcí predikce podle neterminálu a prvního symbolu nezpracované části vstupu. Predikcí dostaneme pravidlo gramatiky, kterým expandujeme neterminál. Nyní nalezneme první nedokončenou část derivačního stromu zleva. Jde-li o terminál, odstraníme jej ze vstupu a připojíme ho jako list do derivačního stromu. Jde-li o neterminál, zjistíme predikci a provedeme jeho expanzi. V tomto postupu pokračujeme, dokud nedostaneme kompletní derivační strom. Pokud se v některém místě nemůže predikce rozhodnou mezi více pravidly, máme špatnou gramatiku, která není LL(1). Pokud v některém místě predikce neexistuje anebo jsme nepřečetli celý vstup, tak řetězec nepatří do dané gramatiky.

Podle představeného průběhu zpracování je možné vidět, že lze celou metodu zobecnit do neměnného algoritmu, který pouze dostane specifickou predikci určenou pro aktuální gramatiku. Tato predikce se vyjadřuje ve formě rozkladové tabulky a její konstrukce představuje hlavní část všech LL metod. U běžného LL(1) parsingu existuje několik rozdílných způsobů tvoření rozkladové tabulky a to v závislosti na tom, jaká pravidla gramatiky povolíme na vstupu. My budeme uvažovat nejobecnější a nejběžnější způsob, který mimo jiné povoluje ε-pravidla.

Tvorba rozkladové tabulky bude prezentována na této gramatice:

$$\begin{aligned} S &\rightarrow a\,A\,b \mid b\,A\,a \\ A &\rightarrow c\,S \mid \varepsilon \end{aligned}$$

Zápis se nijak neliší od běžných zvyklostí. Gramatika je prezentována výčtem jejich pravidel, velká písmena představují neterminály, malá terminály a S je počáteční neterminál. Jde o velmi jednoduchou gramatiku, ale lze na ní dobře prezentovat rozdíly v navazujících pokročilých metodách a zároveň výpočet její rozkladové tabulky zabere rozumné množství prostoru. U složitějších gramatik se přitom postup výpočtu žádným způsobem nezmění, pouze naroste velikost zpracovávaných struktur.

2.2 Funkce First a množiny First

Než začneme tvořit samotnou rozkladovou tabulku, připravíme si nejdříve některé funkce a množiny, které jsou pro tvorbu tabulky nezbytné. Protože budeme k predikci využívat první symbol řetězce, bude se nám hodit funkce $First(w)$, která nám vrátí množinu symbolů, které se mohou objevit jako první v řetězci w. Funkce funguje následujícím způsobem. Pro ε je výsledná množina prázdná. Jinak se vezme množina prvního symbolu z w a přidá se do výsledku. Obsahuje-li množina výsledku ε a zpracovaný symbol nebyl poslední, tak ε z výsledku odstraníme a přidáme do výsledku množinu dalšího symbolu. Takto pokračujeme, dokud funkce neskončí, tj. dokud nenarazíme na symbol, jehož množina neobsahuje ε, nebo neprojdeme celý řetězec w.

Aby tato funkce fungovala správně, je nutné nejdříve vypočítat množiny $First(X)$ pro všechny symboly X dané gramatiky. Začneme s počátečními hodnotami, kdy všechny terminály mají ve své množině sami sebe a všechny množiny pro neterminály jsou prázdné. Jsme tedy v následujícím stavu:

$First(a)$	$First(b)$	$First(c)$	$First(S)$	$First(A)$
$\{a\}$	$\{b\}$	$\{c\}$	$\{\}$	$\{\}$

Nyní necháme proběhnout algoritmus, který z počátečních hodnot vypočítá tranzitivní uzávěr pro množiny neterminálů. V jednom kroku tento algoritmus vždy nachystá nové množiny pro neterminály, projde všechna pravidla gramatiky $X \to \alpha$ a do nové množiny $First(X)$ vždy přidá výsledek funkce $First(\alpha)$. Po první iteraci je tak stav následující:

$First(a)$	$First(b)$	$First(c)$	$First(S)$	$First(A)$
$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{c, \varepsilon\}$

Jelikož algoritmus počítá tranzitivní uzávěr, může trvat několik iterací, než se všechny symboly dostanou na svá místa. Každou další iteraci tak počítáme s výsledky předchozí iterace a algoritmus končí, až se oproti předchozí iteraci nové množiny nijak nezmění. V našem případě stačí provést pouze dvě iterace, protože v druhé se již nic nezmění a předchozí tabulka je tak konečný výsledek. Nyní máme úspěšně připravenou funkci $First(w)$.

2.3 Množiny Follow

Kvůli ε-pravidlům se můžeme dostat do situace, kdy expanzí neterminálu nevznikne žádný terminál. V takovém případě nám k predikci nestačí pouze funkce $First$, ale potřebujeme mít povědomí i o tom, jaké terminály se mohou vyskytnout za daným neterminálem. K tomuto účelu slouží množiny $Follow$ počítané pro všechny neterminály. Aby se s predikcí lépe pracovalo a šlo jednoduše rozpoznat konec vstupního řetězce, přidává se na konec vstupu symbol \$. Tento symbol původně nepatří do gramatiky a slouží jako zarážka,

za kterou již neprobíhá další čtení. Na konec vstup se vkládá tolik zarážek, kolik symbolů používá metoda k predikci. V případě LL(1) parsingu tedy pouze jedna zarážka (symbol $).

Algoritmus pro výpočet množin $Follow$ opět počítá tranzitivní uzávěr a jeho jednotlivé kroky jsou tak podobné výpočtu množin $First$, tj. nachystáme počáteční množiny a iterujeme dokud se obsah množin neustálí. Začneme přípravou počátečních množin, ty budou prázdné a pouze do množiny počátečního neterminálu přidáme zarážku. Jsme tedy v následujícím stavu:

$Follow(S)$	$Follow(A)$
$\{\$\}$	$\{\}$

Nyní začneme s iteracemi. Hledáme všechny tvary pravidel $Y \to \alpha X \beta$, kde α a β jsou libovolný (i prázdný) řetězec a X je neterminál. Pro každý takovýto tvar přidáme do nové množiny $Follow(X)$ výsledek $First(\beta)$, kromě ε, a pokud $First(\beta)$ obsahuje ε, tak ještě také $Follow(Y)$. Jinými slovy, do $Follow$ neterminálu přidáme vše, co za ním může v daném tvaru pravidla následovat anebo také i to, co může následovat za levou stranou tohoto pravidla. Po první iteraci je tak stav následující:

$Follow(S)$	$Follow(A)$
$\{\$\}$	$\{a, b\}$

Po druhé iteraci se ještě množina $Follow(A)$ promítne zpět do $Follow(S)$:

$Follow(S)$	$Follow(A)$
$\{\$, a, b\}$	$\{a, b\}$

Ve třetí iteraci se již nic nezmění a předchozí tabulka je tak konečná.

2.4 Rozkladová tabulka

Nyní již máme připraveny všechny pomocné prvky a můžeme se pustit do tvorby samotné rozkladové tabulky. Rozkladovou tabulku M můžeme indexovat jako dvourozměrné pole $M[X, a]$, kde X je neterminál a a je terminál. Pro usnadnění rozpoznání konce vstupu uvažujeme mezi terminály již zmíněnou zarážku $. Nově pak uvažujeme i zarážku # mezi neterminály, ta představuje konec námi generované větné formy a také původně nepatří do gramatiky. Její použití nám zjednoduší detekci situací, kdy je již sestaven celý derivační strom. Prázdná tabulka tedy vypadá takto:

	a	b	c	$
S				
A				
#				

Algoritmus vyplnění tabulky pak funguje následujícím způsobem. Projdou se všechna pravidla gramatiky $X \to \alpha$ a na indexy rozkladové tabulky $M[X, a]$ se přidá záznam α. Znak a reprezentuje všechny symboly z množiny výsledku funkce $First(\alpha)$ a pokud tato množina obsahovala ε, tak také všechny symboly z množiny $Follow(X)$. Na index $M[\#, \$]$ pak ještě přijde záznam *accept*, který představuje úspěšné zpracování vstupu.

Začneme-li zpracováním pravidla $S \to a\ A\ b$, bude $First(a\ A\ b) = \{a\}$ a obsah tabulky se změní následovně:

	a	b	c	\$
S	$a\ A\ b$			
A				
$\#$				

Po zpracování všech zbylých pravidel a přidání záznamu *accept* vznikne následující výsledná rozkladová tabulka:

	a	b	c	\$
S	$a\ A\ b$	$b\ A\ a$		
A	ε	ε	$c\ S$	
$\#$				*accept*

Predikce s rozkladovou tabulku pak probíhá následovně. Jsme u zpracování neterminálu X a nezpracovaná část vstupu začíná symbolem a. Podíváme se na index v rozkladové tabulce $M[X, a]$. Pokud je zde záznam s řetězcem, tak aktuální neterminál expandujeme podle tohoto záznamu. Je-li na daném indexu záznam *accept*, tak je syntaktická analýza kompletní. Pokud se na daném indexu nenachází žádný záznam, tak jde o chybu, kdy vstup nepatří do dané gramatiky.

2.5 Konflikty v rozkladové tabulce

Konstrukce rozkladové tabulky v předchozím příkladu proběhla bez problémů, prováděli jsem ji s LL(1) gramatikou. Pokud použitá gramatika není LL(1), budou v tabulce buňky obsahující několik záznamů. V takovém místě pak nelze běžnou LL(1) metodou deterministicky určit, jaká expanze má být provedena. Demonstrujme si tuto situaci na ukázkové LL(2) gramatice:

$$\begin{aligned} S &\to a\ A\ a\ a \mid b\ A\ b\ a \\ A &\to b \mid \varepsilon \end{aligned}$$

Pro kterou předchozí postup vytvoří následující rozkladovou tabulku:

	a	b	\$
S	$a\ A\ a\ a$	$b\ A\ b\ a$	
A	ε	$b \mid \varepsilon$	
$\#$			*accept*

Pokud takováto situace nastane, existuje několik možností, jak se s problémem vypořádat. Chceme-li zachovat použití LL(1) metody a zároveň máme volnou ruku nad tím, jak vypadají pravidla gramatiky, můžeme se pokusit upravit tvar těchto pravidel. Uvnitř pravidel existují některé typické konstrukce, které znemožňují vytvoření deterministické LL(1) rozkladové tabulky. Těchto konstrukcí se lze často zbavit a přitom zachovat stejný jazyk generovaný danou gramatikou. Bližší popis pro provádění takovýchto transformací lze nalézt například v knize [5], zde se spokojíme pouze z krátkým výčtem v podobě seznamu:

- *Odstranění levé rekurze* – Ať už se jedná o rekurzi přímou nebo nepřímou. Pokud se expanzí pravidel dostaneme znovu ke stejnému neterminálu, bez vygenerování jediného symbolu před ním, tak zákonitě musí existovat více pravidel se stejnou množinou First. Můžeme se proto pokusit levé rekurze zbavit.

- *Faktorizace pravidel* – Máme-li několik pravidel pro stejný neterminál a tyto pravidla začínají stejným terminálem, opět je jasné, že budou mít stejnou množinu First. Tohoto problému se můžeme pokusit zbavit faktorizací.

- *Eliminace pravidel* – Někdy se lze vyhnout konfliktům eliminací některých neterminálů. Takové neterminály expandujeme přímo do pravidel a původní pravidla můžeme zrušit.

- *Redukce množin Follow* – Je-li některá z množin Follow příliš velká a způsobuje tak kolizi s množinami First, můžeme se pokusit přidat nový neterminál, který její velikost zmenší. V ideálním případě pak bude Follow množina u ε-pravidla disjunktní s množinami First ostatních pravidel stejného neterminálu.

Ne vždy lze pomocí předchozích postupů transformovat gramatiku na LL(1). Někdy gramatika převést nelze, někdy máme gramatiku pevně dánu anebo potřebujeme přiřadit sémantiku přesně k určitým pravidlům, která by se transformací ztratila. V takovém případě musíme přejít k jinému řešení. Jednou z možností je vypořádat se s kolizí ve vlastní režii až za běhu překladače, tehdy máme více informací o aktuální sémantice a můžeme se lépe rozhodnout. Teoreticky je i možné napevno zvolit jen jedno pravidlo, tím ale měníme gramatiku, což ve většině případů nebude vyhovující. Nejpraktičtější možností však pravděpodobně bude zvolení jiné metody parsingu, buď některé varianty LR anebo třeba některé pokročilejší varianty LL.

2.6 Způsoby implementace

Existují dva běžné postupy, jak se celá LL(1) metoda převádí do výsledného programu. První varianta nemá žádnou unifikovanou část a celý program

je vystavěn podle konkrétní gramatiky. Druhá varianta poté odpovídá popisu ze začátku kapitoly, kdy má program unifikované řízení a na základě gramatiky je dodána pouze specifická rozkladová tabulka.

Rekurzivní sestup

Jde o pěknou ukázkou využití rekurzivního zanořování funkcí. Celý postup analýzy je založen na vzájemném volání skupiny několika málo funkcí, kdy první volaná funkce na konci vrátí kompletní výsledek.

Pro vytvoření programu tímto postupem budeme potřebovat připravenou rozkladovou tabulku. Nachystáme se funkci pro každý neterminál v indexu tabulky a místo zarážky # vytvoříme počáteční funkci zpracovávající celý vstup i se zarážkou $S\$$. Každá funkce je pak zodpovědná za všechna pravidla svého neterminálu a funguje následovně. Je-li to nutné, podívá se na první nepřečtený symbol, aby určila, které pravidlo se bude zpracovávat. Začne se zpracováním řetězce pravé části pravidla symbol po symbolu. Je-li to terminál, přečte ho ze vstupu. Je-li to neterminál, zavolá funkci pro daný neterminál. Prošla-li funkce všechny symboly, úspěšně končí. Pokud symboly na vstupu nesouhlasí s předpokládanými, jde o chybu.

I když je takovýto program jiný pro každou gramatiku, dá se celý automaticky vygenerovat podle rozkladové tabulky. Ručně je poté možné velmi jednoduše doplnit sémantické akce k libovolným částem pravidel. Derivační strom lze generovat postupně a předávat v návratových hodnotách funkce.

Prediktivní analýza

Tento postup je nerekurzivní a založen na zásobníkovém automatu, který obstarává unifikované řízení. Rozkladová tabulka se zde ještě v prvním indexu doplňuje o zbylé symboly gramatiky, ke kterým přidáváme akci *pop*:

	a	b	c	\$
S	$a\ A\ b$	$b\ A\ a$		
A	ε	ε	$c\ S$	
a	*pop*			
b		*pop*		
c			*pop*	
#				*accept*

Automat začne se zásobníkem obsahujícím zarážku # a neterminál S a na vstupu má analyzovaný řetězec. V každém kroku se pak automat podívá do rozkladové tabulky na index $M[b, a]$, kde b je symbol na vrcholu zásobníku a a je první nezpracovaný symbol vstupu. Při záznamu s řetězcem expanduje neterminál na vrcholu zásobníku. Při záznamu *pop* odstraní vrchol zásobníku a přejde na další symbol vstupu. Záznam *accept* značí úspěšný konec analýzy. A při přístupu na index bez záznamu se jedná o chybu vstupu.

Kapitola 3

Plný LL(1) parsing

Při běžném LL(1) parsingu určujeme další krok na základě aktuálního neterminálu a prvního nezpracovaného symbolu vstupu. Tímto postupem však zanedbáváme jednu z možných informací. Informaci o tom, jak jsme se k aktuálnímu neterminálu dostali. Pokud pravidla daného neterminálu nemohou vygenerovat ε, tak nevzniká žádný problém. Všechny možnosti pro další symbol nalezneme v množinách $First$ a ty okolní kontext nijak neovlivní. Pokud ale některé pravidlo ε generuje, musíme použít množinu $Follow$, které se již zmíněný problém dotýká. Když jsme totiž tyto množiny generovali, přidávali jsme do nich všechny symboly, které se někdy mohou za daným neterminálem vyskytnout. To ale neznamená, že je to stále možné i v rámci konkrétního kontextu. Množiny $Follow$ jsou tak pouze aproximací možností dalšího postupu a při predikci s nimi o správnosti jen doufáme. Shrnutí následků způsobených takovouto neurčitostí naleznete na konci kapitoly, kde jsou detailně porovnány výsledky běžného a plného LL(1) parsingu.

Nyní se zaměřme na popis způsobu, jak s informacemi o okolním kontextu pracovat. Metoda plného LL(1) parsingu k tomu využívá neterminály. Myšlena je taková, že při vytváření rozkladové tabulky mírně modifikujeme gramatiku a množiny možných následujících symbolů uložíme přímo do neterminálů. Dále pak ještě upravíme pravidla gramatiky tak, aby se tyto nové neterminály správně generovaly. Místo počátečního neterminálu S z původní gramatiky pak budeme mít např. neterminál $[S, \{\$\}]$. Ten značí, že jsem v neterminálu S, za kterým může následovat zarážka \$.

Dále popisovaný algoritmus je kombinací postupů z knih [3, 1], aby mohl být posléze jednoduše rozšířen pro potřeby LL(k) metody. Výsledkem tohoto postupu je pouze jinak vygenerovaná rozkladová tabulka, která má již všechny nové informace zakomponované v sobě. Ostatní části parsingu tak zůstávají oproti předchozí metodě nezměněné. Přesnější predikce plného LL(1) parsingu se kromě složitějšího výpočtu rozkladové tabulky projeví také na její velikosti. Každý neterminál se nyní může objevit v několika řádcích tabulky a to vždy s rozdílnou množinou možných následujících symbolů.

3.1 Pomocné tabulky

Než se pustíme do vytváření samotné rozkladové tabulky, připravíme si skupinu pomocných tabulek, v literatuře označovaných jako LL(k) tabulky. Tuto část si lze představit jako náhradu za výpočty $Follow$ v předchozí metodě. Každá pomocná tabulka náleží k některému novému neterminálu a uchovává informace o možném následujícím postupu.

Opět uvažujme stejnou formální gramatiku:

$$S \rightarrow a\,A\,b \mid b\,A\,a$$
$$A \rightarrow c\,S \mid \varepsilon$$

Výpočet pomocných tabulek začne z nového startujícího neterminálu $[S, \{\$\}]$. O něm víme, že určitě musí existovat, protože zarážku \$ vždy přidáváme na konec vstupu a celý vstup musí být vygenerován z původního počátečního neterminálu. V pomocné tabulce poté vyplňujeme sloupce Next, Production a Follow. Nejdříve začněme tím, že projdeme všechna pravidla neterminálu S a pro každé pravidlo vytvoříme v tabulce nový řádek. Do sloupce Production vždy uložíme pravou stranu pravidla a do sloupce Next doplníme množinu vyhovujících symbolů. Tato množina se počítá podobně, jako se v předchozí metodě určovalo vyplňování rozkladové tabulky. Do množiny Next budou patřit všechny symboly z výsledku $First$ pravé strany pravidla a pokud tento výsledek obsahoval ε, tak také všechny symboly z množiny přenášené aktuálním neterminálem:

Table $[S, \{\$\}]$

Next	Production	Follow
$\{a\}$	$a\,A\,b$	
$\{b\}$	$b\,A\,a$	

Nyní zbývá dopočítat sloupec Follow. To se provede následovně. Pro neterminál $[Y, M]$ hledáme postupně všechny tvary pravidla $Y \rightarrow \alpha X \beta$, kde α a β jsou libovolné řetězce a X je neterminál. Pro každý takovýto tvar přidáme do seznamu ve sloupci Follow nový neterminál $[X, N]$, kde množina N obsahuje všechny symboly z $First(\beta)$ a pokud $First(\beta)$ obsahovalo ε, tak také všechny symboly z množiny M:

Table $[S, \{\$\}]$

Next	Production	Follow
$\{a\}$	$a\,A\,b$	$[A, \{b\}]$
$\{b\}$	$b\,A\,a$	$[A, \{a\}]$

Tím máme hotovou první tabulku. Algoritmus dále postupně sbírá všechny nové neterminály vytvořené v rámci sloupce Follow a stejným způsobem pro ně vytvoří další pomocné tabulky. Předchozího tabulka vytvořila dva nové neterminály $[A, \{b\}]$ a $[A, \{a\}]$, pro ty vzniknou tyto pomocné tabulky:

Table $[A, \{b\}]$

Next	Production	Follow
$\{c\}$	$c\ S$	$[S, \{b\}]$
$\{b\}$	ε	–

Table $[A, \{a\}]$

Next	Production	Follow
$\{c\}$	$c\ S$	$[S, \{a\}]$
$\{a\}$	ε	–

Nyní opět dostáváme dva nové neterminály $[S, \{b\}]$ a $[S, \{a\}]$ a opět pro ně vytvoříme další dvě pomocné tabulky:

Table $[S, \{b\}]$

Next	Production	Follow
$\{a\}$	$a\ A\ b$	$[A, \{b\}]$
$\{b\}$	$b\ A\ a$	$[A, \{a\}]$

Table $[S, \{a\}]$

Next	Production	Follow
$\{a\}$	$a\ A\ b$	$[A, \{b\}]$
$\{b\}$	$b\ A\ a$	$[A, \{a\}]$

Zde již nový neterminál nevznikl a generování pomocných tabulek končí.

3.2 Rozkladová tabulka

S nachystanými pomocnými tabulkami je již vytvoření rozkladové tabulky přímočaré. Opět uvažujeme rozkladovou tabulku $M[X, a]$, kde X představuje nové neterminály a zarážku # a a představuje terminály a zarážku \$. Záznamy *accept* a *pop* se do tabulky přidávají původním způsobem. Novým způsobem se pak pouze doplní záznamy pro expanzi neterminálů.

Začneme postupně procházet jednotlivé pomocné tabulky neterminálů X. V každé takovéto tabulce projdeme všechny řádky a následně popsané vkládání opakujeme pro každý symbol a z množiny Next. Na index $M[X, a]$ uložíme záznam s řetězcem ze sloupce Production, ve kterém nahradíme všechny výskyty původních neterminálů za jejich nové verze ze seznamu Follow. Když se podíváme např. na pomocnou tabulku neterminálu $[S, \{b\}]$, dostáváme z ní dva nové záznamy. Na pozici $M[[S, \{b\}], a]$ přidáváme řetězec $a\ [A, \{b\}]\ b$ a na pozici $M[[S, \{b\}], b]$ přidáváme záznam $b\ [A, \{a\}]\ a$. Po doplnění všech záznamů tak získáme následující rozkladovou tabulku:

	a	b	c	\$
$[S, \{\$\}]$	$a\ [A, \{b\}]\ b$	$b\ [A, \{a\}]\ a$		
$[S, \{a\}]$	$a\ [A, \{b\}]\ b$	$b\ [A, \{a\}]\ a$		
$[S, \{b\}]$	$a\ [A, \{b\}]\ b$	$b\ [A, \{a\}]\ a$		
$[A, \{a\}]$	ε		$c\ [S, \{a\}]$	
$[A, \{b\}]$		ε	$c\ [S, \{b\}]$	
#				*accept*

Způsob predikce a význam jednotlivých záznamů rozkladové tabulky zůstává stejný, pouze nyní považujeme za výchozí neterminál $[S, \{\$\}]$. Když si teď projdeme výslednou tabulku, uvidíme rozdílné závislosti neterminálů na okolním kontextu. Neterminál S můžeme expandovat ve třech rozdílných kontextech, ale žádný z nich neovlivní výsledek. Oproti tomu u neterminálu A máme dva možné kontexty a ty ovlivňují reakci na symboly a a b.

3.3 Porovnání výsledků

Nyní, když máme hotové rozkladové tabulky běžnou i plnou LL(1) metodou, můžeme si je zobrazit vedle sebe a podívat se detailně na vzniklé rozdíly:

	a	b	c	\$
S	$a\ A\ b$	$b\ A\ a$		
A	ε	ε	$c\ S$	
#				*accept*

	a	b	c	\$
$[S, \{\$\}]$	$a\ [A, \{b\}]\ b$	$b\ [A, \{a\}]\ a$		
$[S, \{a\}]$	$a\ [A, \{b\}]\ b$	$b\ [A, \{a\}]\ a$		
$[S, \{b\}]$	$a\ [A, \{b\}]\ b$	$b\ [A, \{a\}]\ a$		
$[A, \{a\}]$	ε		$c\ [S, \{a\}]$	
$[A, \{b\}]$		ε	$c\ [S, \{b\}]$	
#				*accept*

Jak již bylo avizované, tabulka vygenerovaná plnou metodou má výrazně více řádků, i když počet sloupců zatím zůstává stejný.

Zaměřme se nyní blíže na nové neterminály S. Kontexty neterminálu S jsou irelevantní. Nijak neovlivní průběh parsingu a pouze zvyšují komplexnost tabulky. Vítaným vylepšením by proto byla jejich následná redukce:

	a	b	c	\$
$[S]$	$a\ [A, \{b\}]\ b$	$b\ [A, \{a\}]\ a$		
$[A, \{a\}]$	ε		$c\ [S]$	
$[A, \{b\}]$		ε	$c\ [S]$	
#				*accept*

I když se tento krok v knihách často neprovádí, umožní nám vidět další významnou vlastnost obou metod. Obecně je ale s redukcí LL tabulek stejný problém jako u LR parsingu. Aby mohla být redukce provedena, musíme nejdříve v paměti uchovat celou původní tabulku. V případě takto malých příkladů to není problém, ale u větších gramatik a hlavně pak u LL(k) metody s vysokým k narůstá velikost tabulky opravdu rapidně.

Přejděme k neterminálu A. Podle kontextu je v plné tabulce rozděleno použití ε-pravidla do dvou řádků a přítomnost špatného symbolu tak může okamžitě vyvolat chybu. Zbylé záznamy pro symbol c jsou nyní totožné a žádné jiné rozdílné záznamy se navzájem nepřekrývají. Možná se to nemusí na první pohled zdát, ale síla běžného a plného LL(1) parsingu je stejná. Oba zpracují tutéž třídu gramatik. Rozdíly se projeví pouze při reakci na chybný vstupní řetězec. Tato vlastnost byla již v knize [1] detailně dokázána a ve zkratce je myšlenka důkazu následující. Plný parsing ovlivní pouze neterminály, ze kterých lze vygenerovat ε. Každý neterminál má maximálně jedno ε-pravidlo, jinak by nemohlo jít o LL gramatiku. Pokud je symbol

v množině Follow, tak se musí v některém kontextu za tímto neterminálem vyskytnout. Jestli tedy Follow koliduje s First, tak i po rozdělení do více kontextů dojde při některém z nich ke kolizi. Je-li původní predikce podle množiny Follow chybná, tak pouze expandujeme neterminál do podoby, ve které a za kterou se ověřovaný symbol nemůže vyskytnout. Nevyhnutelně tedy při dalším postupu dříve či později skončíme chybou.

Předveďme si průběh obou metod na naší gramatice, kdy se budeme snažit zpracovat chybný vstupní řetězec $a\ a$. Začneme s ukázkou expanze podle tabulky běžného LL(1) parsingu:

$$S \ \rightarrow \ a\,A\,b \ \rightarrow \ a\,b$$

Došli jsme k větné formě $a\ b$, která neodpovídá vstupnímu řetězci a při porovnávání symbolů b a a dojde k chybě. Nyní se podívejme na ukázku expanze podle tabulky plného LL(1) parsingu:

$$[S] \ \rightarrow \ a\,[A, \{b\}]\,b$$

Skončili jsme před expanzí neterminálu A, protože v daném kontextu neexistuje expanze, která by vedla ke správnému řešení.

Výhodou plného parsingu je tedy brzká detekce chyby. Chyba je vždy nahlášena okamžitě, jakmile je to možné. Takováto detekce pak může ulehčit následné zotavení, protože nedojde k žádným zbytečným a nesprávným expanzím. Obecně se ale tato výhoda nepovažuje za dostatečně významnou na to, aby se vyplatilo vytvářet a používat výrazně složitější rozkladové tabulky. Situace se však změní u obecnějšího LL(k) parsingu, kde již s aproximovanými Follow množinami nelze zpracovat celou třídu LL(k) gramatik.

Kapitola 4

LL(k) parsing

Metodu LL(k) parsingu si lze představit jako zobecnění plného LL(1) parsingu probíraného v předchozí kapitole. LL(k) parsing se při predikci již nemusí spokojit pouze s první nezpracovaným symbolem vstupu, ale může jich využít rovnou k, kde $k \geq 1$. Toto zobecnění výrazně zasáhne jak do tvorby, tak i do výsledného rozložení a používání rozkladové tabulky. Množiny symbolů již nebudou při výpočtech dostatečné a budeme muset začít pracovat se složitějšími množinami řetězců. Pro ty již nejsou představeny obecné algoritmy, které by fungovaly bez výjimky na všech gramatikách. Před tvorbou rozkladové tabulky tak budeme provádět i počáteční ověření gramatiky.

Novinkou oproti předchozím metodám bude také nutnost zvolení počítaného k. Neexistuje postup pro nalezení nejmenšího dostatečného k, který by nezahrnoval systematické zkoušení sestrojování rozkladových tabulek pro k od 1 výše. Postup s takovýmto zkoušením nemusí nikdy skončit, pokud pro zadanou gramatiku žádné LL(k) neexistuje. Bylo by sice možné zadat určitý limit k, ale většinou se k volí ručně. Pro pokusné k se nechá vygenerovat rozkladová tabulka a podle výsledku se pak případně zkouší jiná hodnota. S velikostí k totiž velmi rychle stoupá složitost výpočtu a zároveň se relativně málo zvyšuje šance na nalezení nekonfliktní rozkladové tabulky. Myšleno je tím to, že pokud je gramatika vhodná pro zpracování LL(k) metodou, tak je nejpravděpodobnější, že bude stačit $k = 1$. Jen menší část reálných gramatik bude vhodná pro LL(k) metodu a bude vyžadovat $k = 2$ a tak dále. Synteticky můžeme samozřejmě vytvořit nekonečné množství gramatik, které budou vyžadovat konkrétní k, ale ty pak v praxi příliš uplatnění nenajdou.

Použití stejné ukázkové gramatiky jako u předchozích metod by zde ztrácelo smysl. Jednalo se o LL(1) gramatiku a vyšla by tak stejná rozkladová tabulka jako u plné LL(1) metody. Použijeme proto jinou velmi jednoduchou LL(2) gramatiku a při výpočtech budeme uvažovat $k = 2$:

$$S \rightarrow a\,A\,a\,a \mid b\,A\,b\,a$$
$$A \rightarrow b \mid \varepsilon$$

4.1 Množiny řetězců

Popis metody začneme definováním množiny řetězců a definicemi operací nad těmito množinami. Nejedná se o nic neintuitivního nebo složitého, následující popis proto slouží hlavně pro ujasnění pojmů. Řetězce jsou v tomto kontextu složeny z terminálních symbolů gramatiky a obsahují maximálně k symbolů. Určujeme tak délku řetězce v rozmezí 0 až k, kdy délka 0 reprezentuje ε. Množina řetězců je množina obsahující libovolné množství unikátních řetězců. Dva řetězce jsou shodné, pokud mají stejnou délkou a jsou složeny ze stejných symbolů ve stejném pořadí.

Zavedeme binární operátor $\oplus_k$, který lze použít mezi dvěma řetězci nebo množinami řetězců. Při použití s řetězci se provede jejich spojení, definované běžným způsobem, a výsledek tohoto spojení se ořízne na maximální délku k. Při použití s množinami řetězců je funkce operátoru obdobná. Ke každému řetězci z první množiny jsou jednotlivě připojeny všechny řetězce z druhé množiny, výsledná spojení jsou oříznuta na délku k a přidána do množiny výsledku. Pro ilustraci dva krátké příklady:

$$a \oplus_2 bc = ab$$
$$\{a, ab, \varepsilon\} \oplus_2 \{aa, b\} = \{aa, ab, aa, b\}$$

4.2 Výpočet First

Stejně jako u předchozích metod potřebujeme pomocnou funkci, která nám bude zjišťovat možné výskyty počátečních symbolů v zadané větné formě. S předchozí funkcí $First(w)$ se již nevystačíme, protože nyní potřebujeme výsledek ve formě množiny řetězců. K tomuto účelu zavádějí knihy funkci $First_k(w)$, která vrací všechny možné počáteční řetězce do délky k. O praktickém způsobu implementace takovéto funkce však již mlčí. Zde bude prezentován algoritmus rekurzivního procházení neterminálů, který však nedokáže pracovat s gramatikami obsahujícími levou rekurzi.

V případě práce s LL(k) metodou nám ale takovéto omezení nevadí. Pokud totiž gramatika obsahuje levou rekurzi, tak zákonitě nastanou v rozkladové tabulce konflikty a nejde o LL(k) gramatiku. Tuto vlastnost tak můžeme otestovat před začátkem konstrukce rozkladové tabulky a případně i okamžitě skončit neúspěchem. Ověření lze provést následovně. Připravíme si funkci $Empty(X)$, která nám bude určovat, jestli jde z daného neterminálu X vygenerovat ε. Lze ji připravit např. pomocí původní funkce $First(w)$, v jejímž výsledku pro dané neterminály vyhledáme ε. Zatím nepracujeme s řetězci, takže původní funkci můžeme bez problémů použít, její ostatní výsledky nás nezajímají. Nyní můžeme začít hledat levou rekurzi. Projdeme postupně všechny neterminály a všechna jejich pravidla. V pravých stranách pravidel nás vždy zajímají všechny neterminály, před kterými se nemusí nacházet žádný symbol. Do této kategorie spadají i neterminály předcházené

sérií jiných neterminálů splňujících $Empty$. Pokud se nám jejich rekurzivním procházením podaří najít některý původní neterminál, tak jsme odhalili levou rekurzi. Gramatika má omezený počet neterminálů a pravidel, tento algoritmus tedy vždy skončí.

Funkci $First_k(w)$, kde w je libovolný řetězec, definujeme následovně. Každý terminál obsahuje v množině pouze řetězec sama sebe. Tedy $First(x) = \{x\}$, kde x je neterminál. A také $First(\varepsilon) = \{\varepsilon\}$. Pro všechny neterminály probíhá rekurzivní zanořování, kdy procházíme všechna jejich pravidla. Výsledky z jednotlivých pravidel se spojují sjednocením množin. Zpracování $First_k(w)$ pak proběhne takto. Vezme se první symbol řetězce $w = av$ a vypočítá se $First_k(a)$. Pokud jsou všechny řetězce ve výsledku délky k, tak úspěšně končíme. V opačném případě vypočítáme $First_k$ dalšího symbolu, kde bude mít k velikost podle počtu symbolů, které schází aktuálně nejkratšímu řetězci ve výsledku do maxima. Tento další výsledek spojíme s původním pomocí operátoru $\oplus_k$. Pokračujeme tak dlouho, dokud nemají všechny řetězce ve výsledku délku k nebo neprojdeme celý řetězec w. Pokud gramatika neobsahuje levou rekurzi, tak při postupném zanořování bude dříve nebo později docházet ke snižování hledaného k a algoritmus nakonec dojde ke kýženému výsledku.

4.3 Pomocné tabulky

Postup přípravy pomocných tabulek zůstává téměř stejný. Pouze budeme používat funkci $First_k(w)$ a pro spojování množin řetězců využijeme operátor $\oplus_k$. Výpočet začneme z nového startujícího neterminálu $[S, \{\$\$\}]$. Všimněte si, že množina možných následujících řetězců nyní obsahuje \$\$. Pracujeme s $k = 2$ a tak na konec vstupu přidáváme dvě zarážky \$. Projdeme všechna pravidla $S \to \alpha$. Do sloupce Production vyplníme α a do sloupce Next přidáme výsledek $First_k(\alpha) \oplus_k \{\$\$\}$. Dále ještě doplníme sloupec Follow. Postupně hledáme všechny tvary pravidla $S \to \alpha X \beta$, kde α a β jsou libovolné řetězce a X je neterminál. Pro každý takovýto tvar přidáme do seznamu ve sloupci Follow nový neterminál $[X, N]$, kde $N = First_k(\beta) \oplus_k \{\$\$\}$. První pomocná tabulka tedy vypadá následovně:

Table $[S, \{\$\$\}]$

Next	Production	Follow
$\{aa, ab\}$	$a\ A\ a\ a$	$[A, \{aa\}]$
$\{bb\}$	$b\ A\ b\ a$	$[A, \{ba\}]$

Algoritmus nyní opět sbírá všechny nové neterminály vytvořené v rámci sloupce Follow a stejným způsobem pro ně vytváří další pomocné tabulky. Předchozí tabulka vytvořila dva nové neterminály $[A, \{aa\}]$ a $[A, \{ba\}]$, pro které tak vzniknou následující tabulky:

Table $[A, \{aa\}]$

Next	Production	Follow
$\{ba\}$	b	–
$\{aa\}$	ε	–

Table $[A, \{ba\}]$

Next	Production	Follow
$\{bb\}$	b	–
$\{ba\}$	ε	–

Pravidla pro neterminál A neobsahují ve svých pravých stranách žádné další neterminály. Tím pádem budou všechny Follow části prázdné a generování pomocných tabulek úspěšně končí.

4.4 Rozkladová tabulka

Indexování výsledné rozkladové tabulky nyní dozná určitých změn. Opět uvažujme rozkladovou tabulku $M[X, a]$, kde X představuje nové neterminály a zarážku $\#$. Druhý index a ale nyní představuje řetězce délky $k = 2$. Sem budou patřit všechny variace řetězců délky k složené z terminálních symbolů gramatiky a pak také všechny variace kratších délek doplněných o zarážky \$. Tím je zajištěno, že jakýkoliv začátek vstupu délky k bude mít svůj index v rozkladové tabulce. Zarážky se mohou vyskytnout pouze na konci, takže již za nimi jiné symboly neuvažujeme.

Způsob vyplnění rozkladové tabulky podle pomocných tabulek pak již zůstává nezměněn. Projdeme všechny pomocné tabulky, kdy podle jejich neterminálů a řetězců z Next určíme potřebné indexy. Na ty pak uložíme záznamy ze sloupce Production, ve kterých se nahradí původní neterminály za jejich nové verze ze seznamu Follow. Nakonec ještě doplníme záznam *accept*, který přijde na index samých zarážek $M[\#, \$\$]$. Výsledná rozkladová tabulka bude tedy vypadat následovně:

	aa	ab	$a\$$	ba	bb	$b\$$	$\$\$$
$[S, \{\$\$\}]$	$a[A, \{aa\}]aa$	$a[A, \{aa\}]aa$			$b[A, \{aa\}]ba$		
$[A, \{aa\}]$	ε			b			
$[A, \{ba\}]$				ε	b		
$\#$							*accept*

Výchozím neterminálem je $[S, \{\$\$\}]$ a jednotlivé záznamy v tabulce mají stále stejný význam. Při implementaci metody si ale již například nevystačíme s řízením zpracovaným pomocí zásobníkového automatu. Aby mohla být provedena predikce, musíme se podívat na vrchol zásobníku a na první tři symboly nezpracované části vstupu. Podle toho pak upravíme zásobník a případně se posuneme o jeden symbol na vstupu. Takovéto chování neodpovídá klasickému zásobníkovému automatu a je potřeba řešit oklikou. Pokud bychom do tabulky přidávali dříve zmíněné záznamy *pop*, tak se budou nacházet na místech, kde neterminál v prvním indexu odpovídá prvnímu symbolu z řetězce druhého indexu. Ke zpracování neterminálu totiž dochází vždy, když se nachází na vrcholu zásobníku a na začátku vstupu.

4.5 Zhodnocení výsledků

Na námi zpracovávané gramatice lze jednoduše předvést, proč není používání aproximovaných množin Follow při práci s obecným k dostatečné. V minulé kapitole jsme si ukázali, že i když byl neterminál A rozdělen do více řádků, tak spolu žádné záznamy ve stejném sloupci nekolidovaly a při jejich sloučení by nenastal problém. V případě současné tabulky je ale situace odlišná. Podívejme se na neterminál A a sloupec ba. Máme zde dvě odlišné expanze, které se provádějí na základě jiného kontextu. A ke stejnému výsledku lze dojít i při počítání s množinami. Představme si množiny $Follow_k$, kdy pro neterminál A dostáváme $\{aa, ba\}$. Nyní takovouto Follow množinu připojíme k jednotlivým First množinám odpovídajících pravidel. Pro $A \to b$ dostáváme $\{ba, bb\}$ a pro $A \to \varepsilon$ dostáváme $\{aa, ba\}$. Našli jsme tak kolizi při ba, ale z předchozího textu víme, že plná metoda deterministický parsing bez problémů vytvoří. Jádro problému tkví ve spojování řetězců mezi částmi First a Follow, ke kterému při $k = 1$ nedochází.

Rozkladová tabulka se u této metody začala roztahovat i do šířky, při složitějších gramatikách tak můžeme očekávat její velmi rychlé zvětšování. Chování prvního indexu zůstalo stejné jako u plné LL(1) metody, kdy počet řádků závisí na počtu neterminálů a počtu kontextů, ve kterých mohou být neterminály expandovány. Počet možných kontextů ale narůstá se zvětšujícím se k, kdy již nejsme omezeni pouze počtem terminálů, ale počtem všech variací řetězců terminálů délky k. Druhý index pak přímo vychází z počtu variací řetězců terminálů délky k. Kvůli zúčastněným variacím znamená každé zvýšení k stále výraznější nárůst rozkladové tabulky. V reálu tak lze prakticky používat převážně pouze jednociferné hodnoty k. Při testech na mírně modifikované gramatice jazyka C šlo například rozumně dopočítat rozkladovou tabulku pouze pro $k = 4$. Se zvětšujícím se k také velmi rychle stoupá počet buněk, které neobsahují žádné záznamy, což může vést k možnosti velmi efektivní redukce. Nastává zde ovšem již dříve zmíněný problém, kdy při redukci musíme nejdříve pracovat s kompletní rozkladovou tabulkou.

Oproti předchozím metodám je LL(k) parsing obecně silnější a to v závislosti na zvoleném k. Pro počáteční $k = 1$ je síla totožná, každé následné zvýšení k ale umožní zpracovat další nové gramatiky. Možnosti využití jsou nejzajímavější hlavně okolo $k = 2$, kdy se ještě příliš neprojeví zvětšování rozkladové tabulky. Při této hodnotě lze například vyřešit typický problém u gramatik programovacích jazyků, kdy máme token identifikátoru, ale jeho význam závisí až na dalším tokenu. V případě běžné LL metody se tento problém musí řešit oklikou mimo rozkladovou tabulku, kdežto LL(2) rozkladová tabulka by obsahovala řešení přímo v sobě. Nevýhodou sice ještě stále zůstává nutnost vytvoření nového složitějšího systému řízení, ale tento problém odstraní modifikace popsaná v následující kapitole.

Kapitola 5

LL(k) parsing při čtení po jednom symbolu

Modifikace LL(k) metody představená v článku [4] přináší možnost zpracování libovolného k pomocí stále stejného systému řízení založeného na zásobníkovém automatu. Základní myšlenka modifikace spočívá v tom, že nemusíme stále dokola kontrolovat k symbolů na vstupu, ale můžeme každý symbol přečíst pouze jednou. Informaci o aktuální situaci si pak zapamatujeme zvlášť. Původní prediktivní analýza představená v kapitole 2 nevyužívá model zásobníkového automatu v plné šíři, prakticky zcela totiž zanedbává práci se stavy automatu. Běžně při ní zůstáváme celou dobu pouze v jednom stavu a jen při úspěšné zpracování přejdeme do jiného koncového stavu. Představovaná modifikace zužitkovává tuto potlačovanou mechaniku a do stavů si ukládá informaci o prvních k nezpracovaných symbolech. Počet použitých stavů se potom, stejně jako šířka předchozí rozkladové tabulky, odvíjí od počtu variací řetězců terminálů délky k a bude tak vždy konečný.

Pro představení modifikace využijeme gramatiku a rozkladovou tabulku z předchozí kapitoly. Do tabulky ale ještě na začátku doplníme chybějící záznamy *pop*, se kterými budeme také dále pracovat:

	aa	ab	$a\$$	ba	bb	$b\$$	$\$\$$
$[S, \{\$\$\}]$	$a[A, \{aa\}]aa$	$a[A, \{aa\}]aa$			$b[A, \{aa\}]ba$		
$[A, \{aa\}]$	ε			b			
$[A, \{ba\}]$				ε	b		
a	*pop*	*pop*	*pop*				
b				*pop*	*pop*	*pop*	
#							*accept*

5.1 Nové rozdělení rozkladové tabulky

Rozkladová tabulka začíná při zobrazení zabírat příliš mnoho místa a novým rozdělením se ještě o něco zvětší, zredukujeme si proto velikost neterminálů.

Neterminál $[S, \{\$\$\}]$ přejmenuje na S, $[A, \{aa\}]$ na A_1 a $[A, \{ba\}]$ na A_2. V názvech stavů budeme používat řetězce terminálů a pro jejich odlišení od běžných řetězců je označíme dvojtečkami po stranách (např. $:ab:$). V prvním indexu tabulky se nám pak kromě symbolů ze zásobníků objeví také symboly ze vstupu. Abychom tyto situace opět nějak odlišili, tak symboly ze zásobníku necháme v původním tvaru a symboly ze vstupu obalíme do hranatých závorek (např. $[a]$). Nové rozdělení rozkladové tabulky pak bude vypadat následovně a roztříští tabulku do čtyř separátních částí:

	stavy délky $< k$	stavy délky $= k$
symboly zásobníku	prázdné	akce parsingu
symboly vstupu	čtení vstupu	prázdné

Části označené jako prázdné nemají pro fungování metody žádný význam a budou nás tak zajímat pouze dvě zbylé části. Čtení vstupu budeme používat jen v situacích, kdy v aktuálním stavu nemáme uložené všechny informace o nezpracovaných k symbolech vstupu. Se zásobníkem budeme naopak pracovat jen v situacích, kdy všechny informace máme. Automat bude při tomto rozdělení fungovat takovým způsobem, že nejdříve načte k symbolů do stavu a následně začne provádět parsing. V parsingu bude pokračovat tak dlouho, dokud nenastane situace, kdy by měl zpracovat symbol vstupu. Na takovémto místě ale pouze umaže informaci o prvním symbolu ze stavu. Tím přejde do jiného stavu, který již nemá všechny potřebné informace, a následně tak opět dojde ke čtení ze vstupu. Stav automatu tedy plní funkci jakéhosi bufferu pro prvních k nezpracovaných symbolů.

Příprava indexů pro rozložení modifikované tabulky je poměrně přímočará. První index převezmeme z originální tabulky a přidáme do něj ještě jednou všechny terminály a zarážku \$; oboje s novým odlišným označením. Druhý index pak představuje všechny stavy, které jsou vytvořeny z variací řetězců délky $\leq k$ obsahujících terminály a zarážku \$. Zarážka se objeví pouze na koncích řetězců plných délek. Stav řetězce délky 0 označíme $:0:$. Připravená prázdná rozkladová tabulka tak bude vypadat následovně:

	$:0:$	$:a:$	$:b:$	$:aa:$	$:ab:$	$:a\$:$	$:ba:$	$:bb:$	$:b\$:$	$:\$\$:$
S										
A_1										
A_2										
a										
b										
$\#$										
$[a]$										
$[b]$										
$[\$]$										

5.2 Doplnění obsahu

Na pořadí vyplňování jednotlivých indexů v rozkladové tabulce nezáleží a následující kroky tak lze libovolně přeházet. My začneme vyplněním části pro čtení vstupu, která není nijak ovlivněna původní tabulkou, ale jen terminály gramatiky a samozřejmě velikostí k. Cílem je bezezbytku vyplnit všechny indexy pro symboly vstupu $[y]$ a stavy $:x:$, kde délka $x < k$. Vytváříme tím akce, kdy se přečte symbol ze vstupu a informace o jeho přečtení se uloží do stavu. Ve standardních případech tedy na dané indexy přidáme záznamy $:xy:$. Pro zarážku \$ je ale situace odlišná. Představovaná modifikace se snaží systém řízení zcela odstínit od závislosti na velikost k a tak na konec vstupního řetězce přidáváme vždy pouze jednu zarážku. Tuto situaci je třeba reflektovat při přechodech mezi stavy a pokud pracujeme se zarážkou, tak ji v rámci stavu vždy doplňujeme do velikosti k. Simulujeme tím načtení k zarážek, protože víme, že za první zarážkou by se již nacházely pouze další zarážky. Vyplněná část čtení vstupu tak bude vypadat následovně:

	$:0:$	$:a:$	$:b:$
$[a]$	$:a:$	$:aa:$	$:ba:$
$[b]$	$:b:$	$:ab:$	$:bb:$
[\$]	:\$\$:	$:a$\$:	$:b$\$:

Při vyplňování akcí parsingu se provádí transformace záznamů z původní tabulky. Tyto záznamy pak zůstávají na stejných pozicích. Rozdíl v indexování je pouze ten, že to co jsou ve druhém indexu v původní tabulce řetězce terminálů, tak to jsou v nové tabulce stavy vytvořené právě z těchto řetězců terminálů. Záznamy expanze a záznam *accept* přenášíme do nové tabulky bez úprav. Změny se dotknou záznamů *pop*, které nyní mají jiný význam a potřebují znát následující stav. Do každého záznamu *pop* na pozici y a $:yx:$ tak ještě přidáme informaci o následujícím stavu. Ve standardních případech půjde o stav $:x:$. V případě stavů se zarážkou \$ opět platí stejný speciální přístup jako u minulé části, kdy zarážku v rámci stavů opakujeme do velikosti k. Tím jsme již přenesli všechny záznamy a máme hotovou kompletní rozkladovou tabulku podle představované modifikace:

	$:0:$	$:a:$	$:b:$	$:aa:$	$:ab:$	$:a$\$:$	$:ba:$	$:bb:$	$:b$\$:$	:\$\$:
S				aA_1aa	aA_1aa			bA_2ba		
A_1				ε			b			
A_2							ε	b		
a				*pop* $:a:$	*pop* $:b:$	*pop* :\$\$:				
b							*pop* $:a:$	*pop* $:b:$	*pop* :\$\$:	
$\#$										*accept*
$[a]$	$:a:$	$:aa:$	$:ba:$							
$[b]$	$:b:$	$:ab:$	$:bb:$							
[\$]	:\$\$:	$:a$\$:	$:b$\$:							

Syntaktická analýza s touto metodou pak probíhá následovně. Automat začne se zásobníkem obsahujícím zarážku # a neterminál S, se stavem :0: a se vstupním řetězcem ukončeným jednou zarážkou $. Pokud je ve stavu s řetězcem $< k$, tak hledá predikci podle prvního nezpracovaného symbolu vstupu a aktuálního stavu. Pokud je ve stavu s řetězcem $= k$, tak hledá predikci podle symbolu na vrcholu zásobníku a aktuálního stavu. Při záznamu se stavem přečte symbol ze vstupu a přechází do daného stavu. Při záznamu s řetězcem expanduje neterminál na vrcholu zásobníku. Při záznamu *pop* odstraní vrchol zásobníku a přejde do naznačeného stavu. Záznam *accept* značí úspěšný konec analýzy. A indexy bez záznamu opět představují chybu.

5.3 Zhodnocení

Představený systém řízení se může na první pohled zdát výrazně složitější, ale i když mají některé záznamy upravený význam, tak stále můžeme celou rozkladovou tabulku zkonvertovat do pravidel zásobníkového automatu. Menší přehlednost způsobují nově používané stavy, na které nebylo tabulkové zobrazení původně navrženo. Zásobníkový automat ale může měnit stav v každém pravidle a výběr mezi částmi pro čtení vstupu a pro provádění parsingu se vyřeší jednoduše automaticky tím, že v rámci každého stavu máme vždy dostupné pouze akce z jedné konkrétní části. Tomu také zpětně odpovídají dvě nevyužívané části rozkladové tabulky.

Výsledkem je metoda syntaktické analýzy, kterou lze bez modifikace v řízení využít na libovolnou LL(k) gramatiku a ve které se vždy mění pouze rozkladová tabulka. Převod jiných rozkladových tabulek do požadovaného tvaru je přitom poměrné přímočarý a nepředstavuje velkou výpočetní zátěž. Metodu tak lze v praxi použít na libovolnou gramatiku, pro kterou se nám podaří rozumně vypočítat výchozí rozkladovou tabulku. Při vytváření tedy narážíme pouze na stejná omezení, která již byla popsána u předchozích metod. Simulace bufferu vstupního řetězce v rámci stavů nám zapříčiní zvětšení rozkladové tabulky, ale to není nikdy tak zásadní jako zvětšení velikosti k. Řízení syntaktické analýzy také ještě provede více kroků, protože přibyly nové mezi-akce posouvající symboly ve stavech, ale celkově by zase měly být jednotlivé kroky jednodušší, protože nekontrolují k symbolů vstupu.

Nové rozložení rozkladové tabulky přináší také nové možnosti pro případnou optimalizaci. Dvě prázdné části tabulky představují poměrně velké množství indexů, které nebudou nikdy využity. Pro snížení paměťových nároků by tak bylo možné použít dvě menší samostatné tabulky, které by separátně uchovávaly akce pro čtení vstupu a akce pro vlastní analýzu. Část zpracovávající čtení vstupu je navíc velmi jednoduchá na výpočet, takže by šla interpretovat i funkcí a její tabulka by se tak nemusela uchovávat vůbec.

Kapitola 6

Závěr

Během předešlého výkladu jsme si nejdříve ukázali běžný LL(1) parsing, tak jak bývá přednášen v základních kurzech formálních jazyků a překladačů. Následně jsme navázali představením plného LL(1) parsingu, který již nepoužívá aproximované Follow množiny a vyžaduje výpočet pomocných tabulek. Výklad pokračoval ukázkou LL(k) parsingu, který dále rozšiřuje predikci na k symbolů vstupu, ale také komplikuje systém řízení syntaktické analýzy. Nakonec jsme si tedy ještě představili modifikaci LL(k) parsingu, která umožňuje zpracovat libovolně velké k pomocí unifikovaného systému řízení založeného na modelu zásobníkového automatu. Běžná LL(1) metoda je jednou z nejpopulárnějších metod syntaktické analýzy a to hlavně kvůli její jednoduché implementaci a jednoduchému výpočtu rozkladové tabulky. Oproti složitějším metodám ale nezpracuje příliš velkou třídu gramatik. Pokročilé LL metody představují zajímavou alternativu, která může mít větší sílu a zároveň si může zachovat jednoduchou implementaci. Hůře na tom bude rozkladová tabulka, která již nepůjde jednoduše vytvořit ručně. S pomocí automatického nástroje ale nemusí tato nevýhoda představovat velkou překážku; stále však musíme udržet velikost rozkladové tabulky v rozumných mezích. I s pokročilými LL metodami ale můžeme narazit na konstrukce uvnitř gramatik (typicky levou rekurzi), které nám zabrání ve vytvoření nekonfliktní rozkladové tabulky při libovolném k. Zde již nejspíš budeme muset přejít na syntaktickou analýzu zdola nahoru a využít některou LR metodu.

Kniha [3] ve svém přehledu zmiňuje ještě dva parsingy, které jsou zařazeny mezi LL metody. Oba se snaží řešit některé problémy, které u předchozích LL metod vznikají, a ve výsledku zpracovávají mírně odlišné třídy gramatik. První metodou je lineárně aproximovaný LL(k) parsing, který rozšiřuje běžný LL(1) parsing pro predikci s k symboly. Ten pak má k menších rozkladových tabulek místo jedné velké. Druhou metodou je LL-regulární parsing, který zavádí neomezeně velký pohled na nezpracovanou část vstupu a to v situacích, kdy je to potřeba. Toho dosahuje tak, že za použití heuristik aproximuje původní gramatiku regulárními gramatikami.

Literatura

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing.* Prentice Hall, Inc., 1972. ISBN: 0-13-914556-7.

[2] Milan Češka, Tomáš Hruška, and Miroslav Beneš. *Skripta: Překladače.* 1999.

[3] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide.* Springer, second edition, 2008. ISBN: 978-0-387-20248-8.

[4] Dušan Kolář. Simulation of llk parsers with wide context by automaton with one-symbol reading head. In *Proceedings of 38th International Conference MOSIS '04 - Modelling and Simulation of Systems*, pages 347–354, 2004.

[5] Alexander Meduna. *Automata and Languages: Theory and Applications.* Springer, London, 2000.