OSTRAVSKÁ UNIVERZITA V OSTRAVĚ

REGULÁRNÍ A BEZKONTEXTOVÉ JAZYKY II

HASHIM HABIBALLA

OSTRAVA 2005

Recenzenti:

RNDr. PaedDr. Eva Volná, PhD.
Mgr. Rostislav Fojtík

Název: Regulární a bezkontextové jazyky II
Autoři: Dr. Hashim Habiballa, PhD.
Vydání: první, 2005
Počet stran: 73
Náklad:
Tisk:

Studijní materiály pro distanční kurz: Regulární a bezkontextové jazyky II

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

Vydavatel a tisk: Ostravská univerzita v Ostravě,

ISBN

Regulární a bezkontextové jazyky II.

Hashim Habiballa

Cíl předmětu

Navázat na kurz Regulární a bezkontextové jazyky I. – pokračování problematiky syntaktické analýzy bezkontextových jazyků. Speciální třídy jazyků pro efektivní analýzu a algoritmy pro jejich realizaci.

Formální jazyky a gramatiky. Syntaktická analýza bezkontextových jazyků metodou „shora-dolů“ a „zdola-nahoru“. Zásobníkové automaty, speciální algoritmy založené na principu zásobníkových automatů. Q-gramatiky, jednoduché LL(1) gramatiky, silné a slabé LL(k) gramatiky. Stručný úvod do LR gramatik, vlastnosti a vztahy LL a LR gramatik a jazyků. Vhodné metody implementace algoritmů (metoda rekurzivního sestupu).

Obsah:

Úvod pro práci s textem pro distanční studium.

Průvodce studiem:

Účel textu

Tento text navazuje na studijní oporu Regulární a bezkontextové jazyky I. Věnuje se především aplikaci poznatků pro syntaktickou analýzu bezkontextových jazyků. Syntaktická analýza bezkontextových jazyků je velmi důležitým problémem při tvorbě překladačů různých typů – ať už jde o překladače programovacích jazyků, programy pro podporu sazby nebo značkovací jazyky. Text je více aplikovaně orientován, i když obsahuje rovněž jisté množství definic a vět, zaměřuje se především na postupy (algoritmy) tvorby a výpočtu syntaktických analyzátorů, ale rovněž konkrétní metody implementace.

Struktura

V textu jsou dodržena následující pravidla:

- je specifikován cíl lekce (tedy co by měl student po jejím absolvování umět, znát, pochopit)
- výklad učiva
- řešené příklady a úlohy k zamyšlení
- důležité pojmy – otázky
- korespondenční úkoly (mohou být sdruženy po více lekcích)

Pokuste se tuto oporu využívat nejen při studiu dalších teoretických témat, ale využijte ji pro prohloubení učiva z prvního dílu opory. K tomu Vám poslouží především software, který máte k dispozici v balíčku studijních materiálů a pomůcek. Na konci textu najdete návod, jak tyto pomůcky používat.

1. Syntaktická analýza bezkontextových jazyků

Cíl:

Po prostudování této kapitoly pochopíte:
- Co je syntaktická analýza (SA)
- Kde se používá v reálných aplikacích
- Proč k zápisu syntaxe používáme bezkontextové jazyky

Naučíte se:

- Vytvořit jednoduchý (naivní) model syntaktického analyzátoru

Při studiu základů teorie formálních jazyků a zejména dvou nejjednodušších tříd jazyků – regulárních a bezkontextových – jste se setkali s duálním konceptem gramatiky a automatu k danému jazyku. Gramatika umožňuje generovat daný jazyk (tedy jednotlivé prvky jazyka) a automat naopak rozpoznávat, zda testovaný prvek patří do jazyka. Z tohoto teoretického pohledu může někdy zůstat v pozadí fakt, že tento duální koncept znáte přímo ze své praxe informatiků. Pravděpodobně nejbližším vám bude příklad z oblasti programování a tedy programovacích jazyků. Pokud se učíte používat daný programovací jazyk, učíte se zejména správně zapisovat programy dle jeho specifikace (odhlédneme-li nyní od toho, že chcete aby program dělal to co požadujete – to je vyšší stupeň). Učíte se tedy správně zapisovat **syntaxi** daného programovacího jazyka (tedy jeho strukturu z hlediska jazykových vyjadřovacích prostředků). Například u jazyka Pascal víte, že musí obsahovat nejprve deklarace typů, proměnných, dále deklarace funkcí a procedur a nakonec samotnou výkonnou část s popisem algoritmu – programu. Nebo na mnohem nižší úrovni víte, že aritmetický výraz vložený do přiřazovacího příkazu se může skládat s podvýrazů vzájemně spojených operátory sčítaní, odčítání apod. Přičemž nejjednodušším operandem může být kupříkladu celé číslo a důležité je, že tyto výrazy se mohou do sebe vzájemně vnořovat, čímž můžete vytvářet potenciálně nekonečně složité vnořené výrazy.

Syntaxe

Toto je vlastně malá část oné syntaxe jazyka, kterou musíte zvládnout. Když uděláme analogii s vašimi teoretickými poznatky z předchozího studia, učíte se vlastně **gramatiku** daného jazyka. Co však s oním duálním konceptem automatu? I on je vaší práci zcela přirozeně přítomen. Po správném napsaní programu samozřejmě vaše práce nekončí. Musíte si zdrojový kód programu pomocí zvoleného **překladače** (kompilátoru) přeložit do formy spustitelného nebo jiného cílového kódu. Součástí každého překladače musí být (mimo jiných mnoha dalších kroků) kontrola, zda je váš program správně zapsán podle specifikace jazyka. Tuto kontrolu

Gramatika

Překladač

Syntaktický analyzátor

musí provést jistá část překladače – algoritmu, která z teoretického hlediska funguje jako **automat**. Dá vám odpověď, zda je váš program správně napsán – tedy zda zdrojový kód patří do jazyka Pascal. Samozřejmě u pokročilých překladačů dostanete daleko více informací, včetně typu případné chyby a místa, kde chyba vznikla. V nejjednodušším případě pouhé kontroly typu ANO/NE (program je správně/není správně syntakticky zapsán) se jedná o takzvanou **syntaktickou analýzu** (dále budeme zkracovat **SA**). V anglicky psaných zdrojích se setkáte spíše s jednoslovným označením „**parsing**“. O daném postupu - algoritmu jak tuto SA provést, pak hovoříme jako **syntaktickém analyzátoru** (anglicky „**parser**“).
Z vašich znalostí rovněž vyplývá, že už znáte poměrně naivní metody, jak tyto analyzátory sestrojit. Jsou jimi například zásobníkové automaty. Problém však je (stejně jako i v jiných problémech informatiky), jak tyto analyzátory sestrojovat tak, aby byly dostatečně efektivní („rychlé“). Je jasné, že překladač Pascalu, který by váš program kompiloval celé hodiny by asi neměl pro vás žádný užitek. Podobně jako u jiných problémů, proto půjde především o to, jak najít efektivní analyzátory. Odpovědí bude jisté zjednodušení a okleštění příliš obecných bezkontextových gramatik a především tím se budeme v celém kurzu zabývat.

1.1. Bezkontextová gramatika a zápis syntaxe jazyka

Bezkontextová gramatika (BKG) je jedním z velmi vhodných způsobu zápisu syntaxe jazyků. Syntaxí zde rozumíme jejich jazykovou strukturu. Umožňuje totiž vyjádřit většinu technik, které například u programovacích jazyků používáme. Jde o alternativu několika možností, opakování stejného jazykového výrazu a hlavně vnořování celých rozvětvených struktur mezi sebou. Poslední zmiňovaná technika je právě onou technikou, kterou neumíme vyjádřit pomocí jazyků regulárních, ale teprve pomocí bezkontextových jazyků. Zkusme si představit velmi omezenou část nějakého programovacího jazyka – například strukturu aritmetického výrazu. Principiálně je většina jiných struktur velmi podobných (např. sekvence příkazů je analogická opakovanému sčítání podvýrazů!).

Řešený příklad 1:

Sestrojme BKG pro jazyk složený z aritmetických výrazů s operandem x, operacemi +,* a umožňující vnořovat další podvýrazy stejného typu pomocí symbolů závorek (,). Kupříkladu se může jednat o výraz:
x * (x + x + x)

Gramatiku sestrojíme hierarchicky – tedy aby byla rozlišena priorita operátorů a využijeme „rekurzivní“ vlastnosti přepisu neterminálu, abychom docílili možnosti generovat opakovaně sčítání a násobení.

$G = (\{S, A, B\}, \{x, *, +, (,)\}, S, P)$

P:
$S \rightarrow_1 A + S$, $S \rightarrow_2 A$ *(opakovaný přepis na S nám umožní generovat libovolně mnoho sčítání struktury A)*
$A \rightarrow_3 B * A$, $A \rightarrow_4 B$ *(opakovaný přepis na A nám umožní generovat libovolně mnoho násobení struktury B)*
$B \rightarrow_5 (S)$, $B \rightarrow_6 x$ *(rekurzivním přepisem na S můžeme vnořit libovolně mnoho podvýrazů zcela stejné struktury jako výraz sám do závorek anebo ukončit generování operandem x)*
(Pozn.: index u symbolu $\rightarrow$ určuje pomocné číslo pravidla)

V této gramatice pak lze snadno generovat například výše uvedený výraz x * (x + x + x):

$S \Rightarrow_2 A \Rightarrow_3 B * A \Rightarrow_6 x * A \Rightarrow_4 x * B \Rightarrow_5 x * (S) \Rightarrow_1 x * (A + S)$
$\Rightarrow_4 x * (B + S) \Rightarrow_6 x * (x + S) \Rightarrow_1 x * (x + A + S)$
$\Rightarrow_4 x * (x + B + S) \Rightarrow_6 x * (x + x + S) \Rightarrow_2 x * (x + x + A)$
$\Rightarrow_4 x * (x + x + B) \Rightarrow_6 x * (x + x + x)$

(Pozn.: index u symbolu $\Rightarrow$ určuje pomocné číslo pravidla)

Vidíte, že daná, poměrně jednoduchá gramatika, dokáže generovat relativně složitou strukturu, jakou je aritmetický výraz z hlediska hierarchie vnoření. Zároveň jsme díky indexům získali informaci o způsobu konstrukce výrazu – sekvence 23645146146246. Tím, že jsme navíc provedli kanonickou derivaci – levé odvození (vždy přepisujeme nejlevější neterminál) směřujeme k jednoznačnému postupu – algoritmu, jak generovat konkrétní výraz. Spolu s intuitivním pravidlem pro výběr varianty 1 nebo 2 resp. 3 nebo 4, které vybírá dle toho, zda je ještě přítomen v požadovaném výrazu další operátor stejného typu nebo ne, nám dává toto levé odvození deterministickou možnost krok po kroku derivovat právě požadovaný výraz. Samozřejmě je ještě potřeba správně vybrat pravidlo 5 nebo 6, ale to lze učinit jednoduše dle toho, zda se vyskytuje jako následující požadovaný znak x nebo (.

Zmiňovaný **determinismus** – tedy schopnost jednoznačně určit, které pravidlo máme použít – není samozřejmě automatický. Jsou gramatiky, kde jej nebudeme schopni splnit, což má poměrně značný dopad na efektivitu takového procesu. Kdybyste nevěděli, které pravidlo si vybrat, pokud máte více možností, museli byste zkoušet v podstatě všechny možnosti, které existují a čekat, zda dojdete k požadovanému výrazu. To obecně vede k takzvané „**kombinatorické explozi**“, což je vytváření obrovského množství možností geometrickou řadou. Takováto exploze samozřejmě značně omezuje použití daného postupu pro reálné aplikace. Je tedy jasné, že vhodný tvar výchozí gramatiky je pro reálné aplikace

zásadní. A taktéž vám asi začíná být zřejmé, že i pro některé bezkontextové jazyky není vůbec možné deterministické a zároveň efektivní postupy najít.

1.2. Backusova-Naurova forma

Backusova-Naurova forma

Dalším přehledným a hlavně v praxi ještě více využívaným způsobem zápisu syntaxe jazyka je takzvaná **Backusova-Naurova forma (BNF)**. Jde o zápis podobný bezkontextové gramatice, ale přitom bližší spíše programátorům, resp. praxi.

BNF obsahuje podobně jako BKG neterminály, které se uvádějí do úhlových závorek a přepisují skrze symbol := na řetězce terminálních a neterminálních symbolů. Jde tedy o pravidla tvaru:

$<X> := \alpha_1 \mid \ldots \mid \alpha_n$

Pro přehlednější zápis je však ještě lepší modifikace BNF zvaná EBNF (Extended BNF) – rozšířená BNF, která zjednodušuje zápis opakovaně používaných, příp. podmíněně vyskytujících se výrazů. Umožňuje následující zápisy:

$\{\alpha\}$ – znamená, že výraz se vyskytuje v libovolném počtu (ekvivalent operace iterace)
$\{\alpha\}_n^m$ - znamená, že výraz se vyskytuje v počtu nejméně n a nejvýše m (ekvivalent operace mocniny od n do m)
$[\alpha]$ – znamená, že výraz se může a nemusí na daném místě vyskytnout - je to ekvivalentní zápisu $\{\alpha\}_0^1$

Řešený příklad 2:

Gramatika z předchozího příkladu by v BNF mohla být zapsána například takto:

<aritmetický výraz+> := <aritmetický výraz*>{+<aritmetický výraz*>}
<aritmetický výraz*> := <operand/podvýraz>{*<operand/podvýraz>}
<operand/podvýraz> := (<aritmetický výraz+>) | x

BNF umožňuje přehledný zápis a navíc i jednoduchý přechod k některým typům SA, které však budeme probírat spíše ve vyšším kurzu Překladače. V závěru textu se ale této metodě SA alespoň okrajově budeme věnovat. S pomocí BNF je zapsána například celá gramatika jazyka Pascal v učebnici [Ji88]. Příkladem může být deklarace podmíněného příkazu:

<podmíněný příkaz> := if <booleovský výraz> then <příkaz>|
if <booleovský výraz> then <příkaz> else <příkaz>

1.3. Syntaktická analýza v reálných aplikacích

Použití syntaktické analýzy v reálných aplikacích už trochu vyplývá z předchozích podkapitol. Jedním ze stěžejních použití je využití SA jako součásti překladače. Překladač je algoritmus (program), který k libovolnému kódu ve zdrojovém jazyce (zdrojový kód) vytvoří kód v cílovém jazyce (cílový kód). Tento proces se skládá z mnoha částí a zejména v počáteční fázi překladu hraje SA významnou roli. Počáteční fáze překladu integruje zejména tři druhy analýzy kódu:

1. Lexikální analýza
2. Syntaktická analýza
3. Sémantická analýza

První část tedy lexikální analýza shlukuje symboly do takzvaných lexikálních elementů. Příkladem takového elementu v programovacím jazyce může být například identifikátor. Typicky lexikální analýza nepřesahuje složitostí úroveň regulárních jazyků. Tu pak obstarává SA, která již pracuje s připravenými lexikálními elementy a vytváří jistou reprezentaci derivačního stromu podle konkrétní gramatiky. Sémantická analýza pak řeší problematiku kontroly určitých vazeb programu jako jsou vazby typů proměnných apod.

Samozřejmě, že překladač nemusí provádět pouze překlad zdrojového kódu v nějakém programovacím jazyce. Existují překladače zdrojových kódů textů v programech pro podporu sazby textu jako je např. TeX, kde popisujete text pomocí „příkazů“. Nebo lze provádět překlad mezi různými formáty např. RichTextFormat vs. TeX.

1.4. Syntaktická analýza „shora dolů“

Analýza „shora dolů“

Zásadní rozdělení přístupů v SA spočívá ve způsobu konstrukce derivačního stromu odvození pro daný jazyk (gramatiku). Prvním přístupem je analýza principem „**shora dolů**“ (anglicky top-down parsing). Tento princip vychází při analýze z myšlenky postupné dopředného odvozování na zásobníku od počátečního neterminálu S a srovnávání analyzovaného slova po terminálních symbolech, které se objeví na zásobníku až dojdeme do situace, kdy nám nezbude již nic ke srovnání a to jak v původním slově, tak v přepisovaných řetězcích od S na zásobníku. Tyto kroky, kdy přepisujeme od S nazýváme **expanze** (jelikož neterminály rozšiřujeme na řetězce podle pravidel).
Podívejme se na příklad takovéto analýzy.

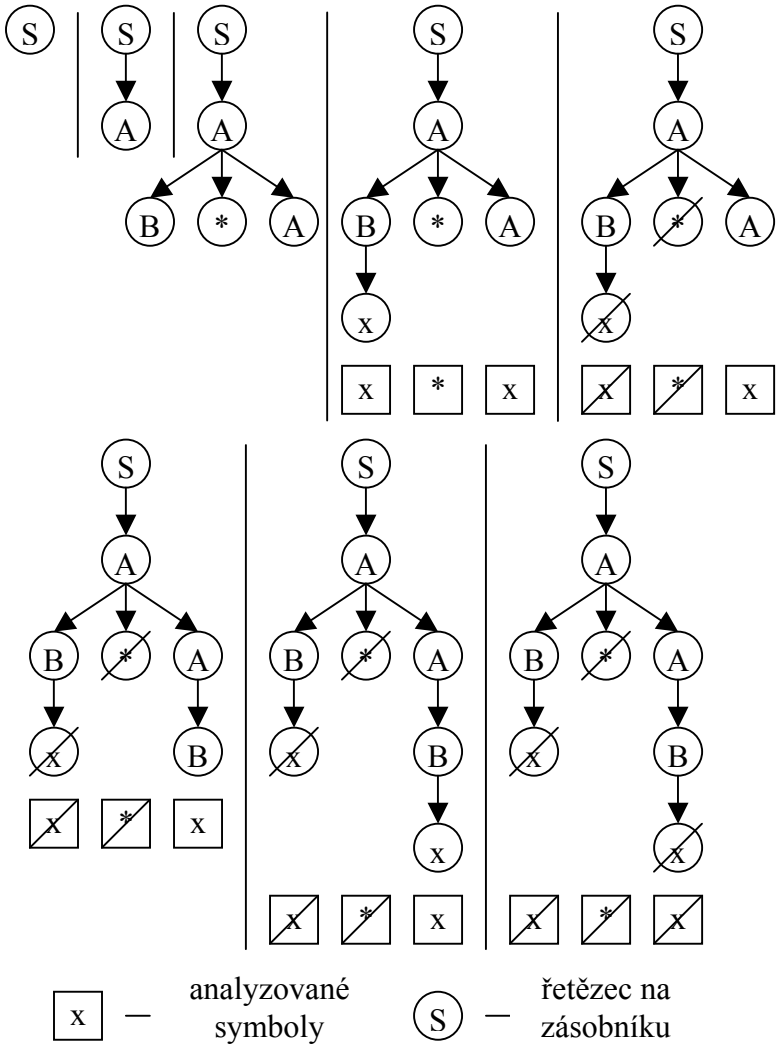
Řešený příklad 3:

Mějme gramatiku dle příkladu 1. G = ({S, A, B},{x, *, +, (,)}, S, P)

P: $S \rightarrow_1 A + S$, $S \rightarrow_2 A$
$A \rightarrow_3 B * A$, $A \rightarrow_4 B$
$B \rightarrow_5 (S)$, $B \rightarrow_6 x$

Postupné expanze a srovnání od S můžeme lineárně a graficky znázornit, přičemž odvození zůstává stejné (každý krok odpovídá kroku ve znázornění, podtrženým písmem a symbolem ≈ zobrazujeme srovnávané terminální symboly). Slovo, které chceme rozpoznat zvolíme pro ilustraci jednoduché:
x * x

$S \Rightarrow_2 A \Rightarrow_3 B * A \Rightarrow_6 x * A \approx \underline{x} * A \approx \underline{x *} A \Rightarrow_4 x * B \Rightarrow_6 \underline{x *} x \approx \underline{x * x}$



1.5. Syntaktická analýza „zdola nahoru“

Druhým přístupem je analýza principem „**zdola nahoru**“ (anglicky bottom-up parsing). Tento princip vychází při analýze z myšlenky postupné zpětné odvozování tím, že postupně vkládáme symboly do

zásobníku a pokud se nám vyskytne v zásobníku řetězec, který se vyskytuje u některého z pravidel na opačné (pravé straně), tak provedeme zpětně odvození na daný neterminál. Princip je tedy zcela opačný – řetězce zjednodušujeme na neterminály. Tomuto opaku expanze se říká **redukce**. Slovo je přijato, pokud dojdeme k situaci, že slovo je celé přečteno a na zásobníku zbyl pouze neterminál S. Podívejme se na příklad takovéto analýzy.

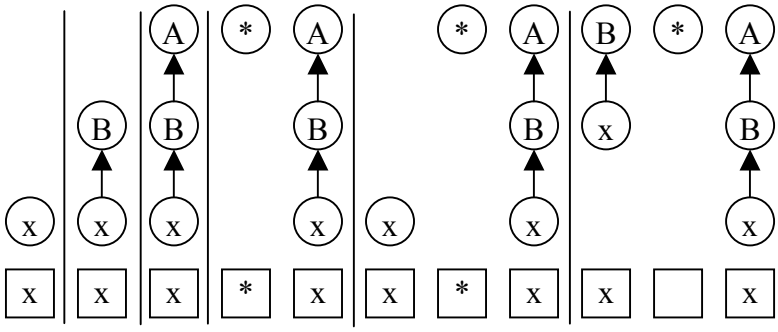
Analýza „zdola nahoru“

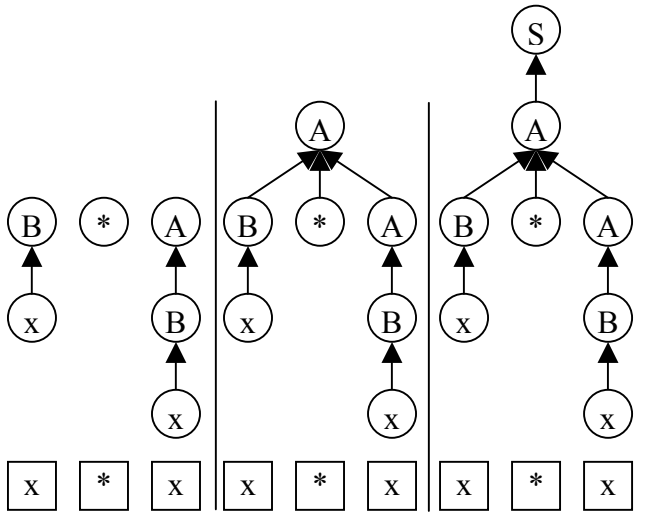
Řešený příklad 4:

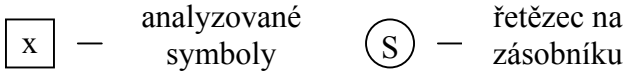
Mějme gramatiku dle příkladu 1.

Postupné redukce a vkládání do zásobníku můžeme lineárně a graficky znázornit, přičemž odvození zůstává stejné (každý krok odpovídá kroku ve znázornění, podtrženým písmem a symbolem ≈ zobrazujeme vkládané terminální symboly). Slovo, které chceme rozpoznat zvolíme pro ilustraci jednoduché:
x * x (Pozor! V tomto případě musíme slovo číst v obráceném pořadí, pokud chceme opět dostat levé odvození.)

$\underline{x} \Leftarrow_6 B \Leftarrow_4 A \approx \underline{*}\ A \approx \underline{x\ *}\ A \Leftarrow_6 B\ \underline{*}\ A \Leftarrow_3 A \Leftarrow_2 S$







Oba principy analýzy bychom mohli zkoumat z pohledu jejich intuitivnosti i efektivity. Pravděpodobně intuitivnější se vám bude zdát SA „shora dolů“, neboť hierarchicky prochází jednotlivé struktury od nejsložitějších k nejjednodušším. Naopak analýza „zdola nahoru“ hledá možné „střípky skládanky“ a postupuje tím méně přehledně k nejvyššímu celku v hierarchii.

V textu si ukážeme oba přístupy na konkrétních typech BKG. Bude se jednat o takzvané **LL a LR gramatiky**. Zkratky LL a LR vyjadřují jednak způsob čtení analyzovaného slova („left-to-right“ – zleva doprava nebo „right-to-left“ naopak) a druhé písmeno pak vyjadřuje typ derivace, který dostaneme analýzou takových gramatik (left – levá derivace, right – pravá derivace).

I způsob využití v praxi souvisí s výše zmiňovanou „intuitivností“. Pokud budete chtít konstruovat analyzátory (překladače) spíše vlastními silami („ručně“), pak použijete přehlednější LL gramatiky (jsou ovšem slabší, pokud jde o vyjadřovací schopnosti – neumí vyjádřit některé jazyky, které LR gramatiky umí). V případě, že budete chtít použít již hotové automatizované nástroje (existuje jich mnoho a budeme se jimi zabývat ve vyšším kurzu překladače), použijete spíše LR gramatiky. Tyto nástroje vám umožní vygenerovat zcela automaticky analyzátor pro gramatiku. Ovšem tyto typy analyzátorů jsou méně přehledné a jejich algoritmická tvorba je mnohem náročnější na pochopení.

Nejdůležitější probrané pojmy:
- syntaxe jazyka a jeho gramatika
- překladač a automat
- syntaktická analýza a syntaktický analyzátor
- determinismus
- Backusova-Naurova forma
- SA „shora dolů“ a „zdola nahoru“

Úkoly k textu:

1. Sestrojte BKG a Backusovu-Naurovu formu pro jazyk výrokových formulí s jediným atomem x a operacemi konjunkce, disjunkce a implikace (s rozlišením priority) a dále s možností vnořit místo atomu podformuli uzavřenou do závorek.
2. Ke gramatice a vybrané formuli (co nejjednodušší) z úkolu 1. proveďte SA „shora dolů“ a „zdola nahoru“.

2. Zásobníkové automaty a vztah k bezkontextovým jazykům

Cíl:

Po prostudování této kapitoly si zopakujete:
- pojem zásobníkového automatu
- vztah ZA k bezkontextovým jazykům
- konstrukci ZA k danému bezkontextovému jazyku
- možnosti a omezení deterministických ZA

Jedním z modelů syntaktické analýzy, který už znáte z předchozího studia je zásobníkový automat. Tato konstrukce obsahuje vstupní pásku se symboly slova, které chceme rozpoznat, množinu stavů, zásobník a přechodovou funkci, umožňuje rozpoznávat i jazyky, které nejsou regulární a jsou bezkontextové, čímž dává postačující výpočetní sílu například k rozpoznávání syntaxe programovacích jazyků.
Problémem této konstrukce je však její příliš velká obecnost – zásobníkový automat je potenciálně nedeterministický a nedeterminismus způsobuje v praxi při jeho simulaci na klasických deterministických strojích enormní časovou složitost (náročnost). Přesto se v této kapitole musíme o této možnosti zmínit a uvidíte na daných příkladech problémy, které musíme v souvislosti se SA řešit.

2.1. Zásobníkový automat jako model SA

Připomeňme si, jak je zásobníkový automat a jeho výpočet definován.

Zásobníkový automat

Definice 1: Zásobníkovým automatem nazveme sedmici (systém určený sedmi parametry)
***M* = (*Q*, Σ, Γ, δ, q_0,Z_0,*F*), kde Q je konečná neprázdná množina stavů, Σ je konečná neprázdná množina vstupních symbolů (abeceda), Γ je konečná neprázdná množina zásobníkových symbolů, $q_0 \in Q$ je počáteční stav, $Z_0 \in \Gamma$ je počáteční zásobníkový symbol, $F \subseteq Q$ je množina koncových stavů a δ je zobrazení množiny $Q\times(\Sigma\cup\{e\})\times\Gamma$ do množiny konečných podmnožin množiny $Q \times\Gamma^*$ (přechodová funkce). ($\delta: Q\times(\Sigma\cup\{e\})\times\Gamma \rightarrow P(Q\times\Gamma^*)$)**

Z definice je patrné, že takto definovaný zásobníkový automat je nedeterministický.

Neformálně význam δ (tj. předpisu chování ZA *M*):
Je-li $\delta(q,a,X) = \{(q_1,\alpha_1),(q_2, \alpha_2), ...,(q_n, \alpha_n)\}$; $q \in Q$, $a \in (\Sigma\cup\{e\})$, $q_i \in Q$,$\alpha_i \in \Gamma^*$, $i \in \{1, 2, ..., n\}$,

X ∈ Γ, potom když *M* má čtecí hlavu na symbolu a, (konečná ŘJ) je ve stavu q a na vrcholu zásobníku je symbol X, může si *M* vybrat jedno i z {1, 2, ..., n } a posunout čtecí hlavu o jeden symbol vpravo, změnit stav řídící jednotky na q_i a symbol X v zásobníku nahradit řetězcem α_i. Speciálně je-li a=e, může *M* provést tzv. e-krok, při kterém nečte a hlava se tudíž neposunuje. Říkáme také, že *M* provedl instrukci (q,a,X) [(δ) || (→)] (q_i,α_i).
Důležitá je i skutečnost, že mohou existovat q ∈ Q, a ∈ (Σ∪{e}), X ∈ Γ tak, že δ(q,a,X)=∅ (v jistých situacích tedy nemůže automat pokračovat ve výpočtu). Při definici konkrétní přechodové funkce budeme definici obrazu pro takovéto vzory ((q,a,X)) vynechávat.

Konfigurace

Definice 2: Mějme ZA *M* = (*Q*,Σ,Γ,δ,q_0,Z_0,*F*). Situací (konfigurací) zásobníkového automatu *M* nazveme trojici (q, w, α), kde q ∈ Q, w ∈ Σ^* a α ∈ Γ^*. q je stav ŘJ, w je slovo (ta část slova) na vstupní pásce, která zbývá přečíst, α je obsah zásobníku. (Nejlevější symbol v α představuje *vrchol zásobníku*). Jestliže (q′, α) ∈ δ(q,a,X), pak pro lib. w ∈ Σ^*, β ∈ Γ^* vede situace (q, aw, Xβ) bezprostředně k situaci (q′,w,αβ), symbolicky značíme:
(q,aw,Xβ) ⇒ (q′,w,αβ)
Nechť E a E′ jsou situace ZA *M*, pak řekneme, že E vede k situaci E′, značíme E $\Rightarrow^*$ E′, jestliže existují situace E_1, E_2,..., E_n tak, že E=E_1⇒ E_2⇒...⇒ E_n=E′. Je-li potřeba, značíme o jaký ZA se jedná:
$\Rightarrow_M$ $\Rightarrow_M^*$

Narozdíl od konečného automatu může ZA rozpoznávat slova nejen tím, že skončí v koncovém stavu, ale také tím, že vyprázdní celý svůj zásobník. Například ilustrace na počátku kapitoly rozpoznává daný jazyk jak prázdným zásobníkem, tak i koncovým stavem.

Rozpoznávání jazyka zásobníkovým automatem budeme definovat dvěma způsoby:
1) přijímání koncovým stavem: slovo w je přijato ZA, jestliže existuje možnost, že po zpracování (přečtení) slova w se automat ocitne v koncovém stavu.
2) přijímání prázdným zásobníkem: slovo w je přijato ZA, jestliže existuje možnost, že po zpracování slova w se ZA ocitne v situaci s prázdným zásobníkem.

Dále víme, že libovolnou gramatiku lze jednoduchým algoritmickým postupem převést na zásobníkový automat, který bude rozpoznávat příslušný bezkontextový jazyk.

Věta 1: Ke každému bezkontextovému jazyku L existuje ZA *M* takový, že L=L_{PZ}(*M*). Navíc *M* má jediný stav.

Důkaz:
Mějme bezkontextovou gramatiku G=(Π,Σ,S,P).
Sestrojíme ZA *M* tak, že L(G)=L_{PZ}(*M*).
Položíme *M* = ({*p*},Σ,Π∪Σ,δ,*p*,*S*,∅).
Pro δ platí:
δ(p,e,X)={(p,α)|(X→ α) ∈ P}; ∀X ∈ Π
δ(p,a,a)={(p,e)}; ∀a ∈ Σ

Takto sestrojený ZA má dva typy pravidel – buď přepisuje neterminál na řetězec nebo srovnává terminální symboly. Pokud symboly nesedí, pak se automat zasekne. Obecně je automat nedeterministický – tedy musí si najít správnou cestu. Pokusme se sestrojit takový automat pro gramatiku z příkladu 1.

Řešený příklad 5:

G = ({S, A, B},{x, *, +, (,)}, S, P)
P: S $\rightarrow_1$ A + S, S $\rightarrow_2$ A
A $\rightarrow_3$ B * A, A $\rightarrow_4$ B
B $\rightarrow_5$ (S), B $\rightarrow_6$ x

Zásobníkový automat sestrojený tímto principem bude následující:
M = ({*p*},Σ,Π∪Σ,δ,*p*,*S*,∅), kde
Σ ={x, *, +, (,)}, Π = {S, A, B}
δ :

1. δ(p,e,S)=(p, A + S)
2. δ(p,e,S)=(p, A)
3. δ(p,e,A)=(p, B * A)
4. δ(p,e,A)=(p, B)
5. δ(p,e,B)=(p, (S))
6. δ(p,e,B)=(p, x)

a dále pravidla pro srovnávání S1. δ(p,x,x)=(p,e), S2. δ(p,*,*)=(p,e), S3. δ(p,+,+)=(p,e), S4. δ(p, (, ( )=(p,e), S5. δ(p,),))=(p,e)

Tento zásobníkový automat můžeme dále využít pro rozpoznávání řetězce (přechody konfigurací jsou ohodnoceny indexem přechodu)
např. x * x

(p, x * x, S) $\Rightarrow_2$ (p, x * x, A) $\Rightarrow_3$ (p, x * x, B * A) $\Rightarrow_6$ (p, x * x, x * A) $\Rightarrow_{S1}$ (p, * x, * A) $\Rightarrow_{S2}$ (p, x, A) $\Rightarrow_4$ (p, x, B) $\Rightarrow_6$ (p, x, x) $\Rightarrow_{S1}$ (p, e , e)

Slovo jsme tedy úspěšně rozpoznali, ovšem analýza proběhla takto hladce, protože jsme do ní zapojili určitou inteligenci. Dokázali jsme „uhodnout“, který přechod použít už například v prvním kroku, kde se nabízel přechod podle 1. i 2. Teoreticky by stroj bez této inteligence musel zkoušet všechny možnosti, vracet se pokaždé zpět ve slově a znovu provádět další možná

odvození. To by samozřejmě byl velmi časově náročný úkol. Jistou naději nám tak může pro efektivní SA poskytnou deterministický ZA.

2.2. Deterministické zásobníkové automaty

Deterministický ZA

Definice 3:
ZA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ nazveme deterministický (DZA), jestliže platí následující dvě podmínky:
1. $\delta(q, a, X)$ je nejvýše jednoprvková množina pro lib. $q \in Q$, $a \in (\Sigma \cup \{e\})$, $X \in \Gamma$.
2. Jestliže $\delta(q,e,X) \neq \emptyset$ pro něj. $q \in Q$, $X \in \Gamma$, pak $\delta(q,a,X)=\emptyset$ pro lib. $a \in \Sigma$.

Definice 4: **Jazyky rozpoznatelné DZA koncovým stavem nazveme deterministické (třídu těchto jazyků označíme Det). Jazyky rozpoznatelné DZA prázdným zásobníkem nazveme bezprefixové deterministické (třídu těchto jazyků označíme BDet).**

Věta 2: **Třída Det je vlastní podtřídou třídy BKJ.**

Tuto vlastnost už sami znáte z předchozí studia. Je jasné, že každý jazyk patřící do Det patří i do BKJ (DZA je pouze speciální případ ZA). Zároveň už víte (i když to nebylo dokázáno), že například jazyk
$L = \{ww^R \mid w \in \{a,b\}^*\}$ není deterministický.
Intuitivně je cítit, že aby mohl tento jazyk rozpoznávat DZA, musel „vědět“, kdy začít symboly ze zásobníku porovnávat s druhou částí slova se zrcadlově obráceným řetězcem. To však při potenciálně nekonečném řetězci není možno univerzálně zajistit.

Pozn. Důkaz této věty by se opíral některé uzávěrové vlastnosti třídy BKJ. Stačilo by ukázat, že ke každému DZA, lze sestrojit DZA, který rozpoznává doplňkový jazyk (podobně jako u deterministického konečného automatu). Jelikož ale třída BKJ není uzavřena vůči operaci doplňku je jasné, že Det a BKJ nemohou být totožné množiny.

Řešený příklad 6:

Příkladem jazyka z Det může být:

$L = \{wcw^R \mid w \in \{a,b\}^*\}$

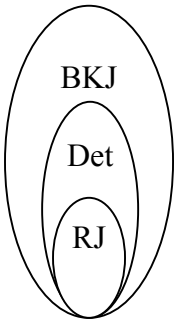
Pro něj jednoduché sestrojit DZA, například takto:

$M = (\{q,r\},\{a,b,c\},\{Z, A, B\}, \delta, q, Z, \emptyset)$

$\delta(q,a,Z)=(q, AZ)$, $\delta(q,b,Z)=(q, BZ)$, $\delta(q,a,A)=(q, AA)$, $\delta(q,b,A)=(q, BA)$,

$\delta(q,a,B)=(q,AB)$, $\delta(q,b,B)=(q,BB)$,
(přechody, které přidávají do zásobníku za každý symbol v první části slova odpovídající zásobníkový symbol)
$\delta(q,c,Z)=(r,Z)$, $\delta(q,c,A)=(r,A)$, $\delta(q,c,B)=(r,B)$,
(přechody určující dosažení poloviny slova, která je signalizovaná pomocí c – nutnost přechodu do nového stavu)
$\delta(r,a,A)=(r,e)$, $\delta(r,b,B)=(r,e)$,
(přechody porovnávající, zda symboly v zrcadlové části odpovídají uloženým symbolů z počátku)
$\delta(r,e,Z)=(r,e)$
(v případě, že vše bylo totožné, rozpoznáváme prázdným zásobníkem)

Deterministické bezkontextové jazyky tedy tvoří jakýsi mezistupeň mezi regulárními a všemi bezkontextovými jazyky. Samozřejmě dokáží popsat vnořování struktur, což je činí mnohem silnějšími než jsou regulární jazyky. V následujícím textu se budeme zabývat ještě omezenějšími třídami deterministických BKJ. Schématicky lze tuto hierarchii popsat:



Hierarchie BKJ

Nejdůležitější probrané pojmy:
- zásobníkový automat - nedeterministický a deterministický
- konstrukce ZA k BKG

Úkol k textu:

1. Sestrojte ZA pro rozpoznávání jazyka výrokových formulí s jediným atomem x a operacemi konjunkce, disjunkce a implikace (s rozlišením priority) a dále s možností vnořit místo atomu podformuli uzavřenou do závorek. Proveďte rozpoznání libovolné formule s alespoň třemi spojkami.

3. Základy syntaktické analýzy „shora dolů“

Cíl:

Po prostudování této kapitoly pochopíte:
- Jaká omezení mají nejjednodušší typy LL gramatik
- Funkci rozkladové tabulky v SA

Naučíte se:
- Vytvářet rozkladovou tabulku pro SLL(1) gramatiku
- Provádět SA pomocí této rozkladové tabulky

Při deterministické syntaktické analýze se v zásadě mohou využívat dva typy informací:

1. Informace o nepřečtené části analyzovaného (vstupního) řetězce
2. Informace o dosavadním průběhu SA

Samozřejmě, že čím méně takovýchto pomocných informací budeme potřebovat, tím jednodušší a přímočařejší SA bude. U LL gramatik se bude využívat především informace o k symbolech, které se v řetězci vyskytují na následujících k pozicích a pouze u obecnějších typů LL gramatik bude potřeba mít k dispozici i informace typu 2.
Z praktického hlediska je to velmi výhodné. Vraťme se opět k programovacím jazykům. Z hlediska konstrukce analyzátoru je velmi pohodlné (a tím i algoritmicky málo složité), když budeme zdrojový kód číst pouze dopředu bez návratů a navíc si nebudeme muset uchovávat nějaké další nadbytečné informace.

3.1. Model analýzy „shora dolů“ a jednoznačnost

Při studiu modelu analýzy typu „shora dolů“ jste viděli, že klíčovým problémem je rozhodnutí, jaké pravidlo použít, pokud máme u jednoho neterminálu více možností na co jej expandovat. Speciální typy gramatik pro tento typ analýzy proto mají omezení, které má především zabránit vzniku nejednoznačnosti při použití přepisovacího pravidla u stejného neterminálu. Podívejme se na následující velmi jednoduchou gramatiku:

Řešený příklad 7:

$G = (\{S, A\},\{a,b,c\}, S, P)$
P: $S \rightarrow_1 aASc$, $S \rightarrow_2 b$
$A \rightarrow_3 a$, $A \rightarrow_4 cSAb$

Lze vygenerovat například slovo: acbabbc
$S \Rightarrow_1 aASc \Rightarrow_4 acSAbSc \Rightarrow_2 acbAbSc \Rightarrow_3 acbabSc \Rightarrow_2 acbabbc$

U této gramatiky je už na první pohled zřejmé, že má pro každý neterminál dvě pravidla. Teoreticky zde tedy hrozí nejednoznačnost při expanzi. Při bližším zkoumání, jakým terminálním symbolem pravidla začínají, ale zjistíme, že každé z pravidel pro určitý neterminál začíná různým symbolem. Nabízí se tedy možnost rozhodnout se podle toho, jaký symbol v analyzovaném slově následuje. U této gramatiky je to poměrně jasné, avšak to jen díky dvěma faktům:

1. Každé pravidlo začíná terminálem – tedy přímo „vidíme“ na jaký symbol máme přepis provést a tedy i vidíme, zda nedochází k přepisu na stejný terminál u dvou pravidel pro stejný neterminál (tzv. **kolize**).
2. Gramatika vůbec neobsahuje pravidlo s e (epsilonem) na pravé straně. Právě e-pravidla by celou situaci ještě mnohem více zkomplikovala, neboť způsobují, že příslušné neterminály mohou (ale nemusí) v odvození „mizet“. To pak v odhalování kolizí způsobuje další nepřehlednost.

Kolize

Jak si později ukážeme je možné jednoznačnou SA provádět i bez splnění těchto podmínek, avšak bude to obtížnější.

3.2. Jednoduché LL(1) gramatiky a rozkladové tabulky

Gramatika z předchozího řešeného případu splňuje podmínky, které předpokládáme u nejjednoduššího typu LL gramatik. Jde o takzvanou **jednoduchou LL(1) gramatiku neboli SLL(1)** (Simple LL). Její základní omezení spočívá v tom, že vždy přepisuje neterminál na řetězec začínající terminálem a dvě pravidla pro jeden neterminál musí začínat různými terminály.

Definice 5: Bezkontextová gramatika G=(Π,Σ,S,P) je jednoduchá LL(1) gramatika nebo SLL(1) gramatika, pokud platí:

1. **$(X \rightarrow a\alpha) \in P$, kde a je terminál a α je řetězec složený z terminálů a neterminálů.**
2. **Když platí $(X \rightarrow a\alpha) \in P$ a $(X \rightarrow b\beta) \in P$, pak $a \neq b$.**

Jednoduchá LL(1) gramatika

Číslo „1“ v názvu SLL(1) určuje počet symbolů, který musíme dopředu znát v analyzovaném slově, abychom byli schopni určit jaké pravidlo použít. Vidíte, že u každého pravidla jsme to schopni určit na základě pouhého jednoho znaku, neboť podmínka 2. vylučuje existenci dvou pravidel pro stejný neterminál začínající stejným znakem.

Pro každou takovou gramatiku (i obecnější typy LL gramatik) je možné sestrojit takzvanou rozkladovou tabulku, která určuje pro každý neterminál a příslušný následující symbol podle jakého pravidla máme provést expanzi. Algoritmus pro vytvoření takovéto rozkladové tabulky pracuje

podle jednoduchého principu – na příslušný řádek (odpovídající neterminálu) a příslušný sloupec (odpovídající vstupnímu symbolu na pravé straně pravidla) se vloží řetězec, který je u zkoumaného pravidla na pravé straně.

Pro gramatiku z předchozího příkladu by rozkladová tabulka vypadala následovně.

Řešený příklad 8:

Mějme gramatiku:
G = ({S, A},{a,b,c}, S, P)
P: S $\rightarrow_1$ aASc, S $\rightarrow_2$ b
A $\rightarrow_3$ a, A $\rightarrow_4$ cSAb

Rozkladová tabulka

M	a	b	c
S	aASc, 1	b, 2	
A	a, 3		cSAb, 4

Máme-li sestrojenou rozkladovou tabulku, můžeme pomocí algoritmu pro syntaktickou analýzu provést rozpoznání daného slova. Tento algoritmus postupně čte vstupní slovo, pracuje s pamětí typu zásobník a provádí 4 možné operace – expanze podle pravidel, porovnání stejných symbolů na zásobníku i ve vstupní řetězci, přijetí slova (pokud se vyprázdní zásobník a slovo je přečteno) a chyba, pokud se dostaneme do situace, pro kterou není v rozkladové tabulce definovaná akce.

Následující formalizace uvedeného postupu je obecným postupem pro LL(1) gramatiky (tedy i pro vyšší typy obecnějších gramatik, které budeme probírat později).

Algoritmus 1: Syntaktická analýza pro LL(1) gramatiky.

Vstup: rozkladová tabulka M pro SLL(1) gramatiku, q-gramatiku nebo LL(1) gramatiku G=(Π,Σ,S,P), vstupní řetězec w $\in \Sigma^*$.

Algoritmus syntaktické analýzy

Výstup: levý rozklad (derivace) vstupního řetězce v případě, že w $\in$ L(G), jinak chybová signalizace.

Postup:

- Algoritmus čte vstupní řetězec, používá zásobník a vytváří výstupní řetězec složený z indexů pravidel.
- Konfigurace je trojice (x, α, π), kde x $\in \Sigma^*$, $\alpha \in (\Pi \cup \Sigma)^*$ a $\pi \in N^*$, kde x je dosud nepřečtená část slova, α je obsah zásobníku a π je posloupnost čísel pravidel reprezentující levý rozklad podle G.

- Počáteční konfigurace je (w, S, e) a algoritmus provádí přechody mezi konfiguracemi podle následujících kroků 1. a 2., dokud nenastane situace 3. nebo 4.
  1. **Expanze:** $(ax, A\alpha, \pi) \Rightarrow (ax, \beta\alpha, \pi i)$, pokud $A \in \Pi$, $M(A, a) = \beta$, i. Symbol A se na vrcholu zásobníku nahradí řetězcem β a číslo i je připojeno k posloupnosti reprezentující levý rozklad.
  2. **Porovnání:** $(ax, a\alpha, \pi) \Rightarrow (x, \alpha, \pi)$, pokud $a \in \Sigma^*$. Totožné symboly na vrcholu zásobníku a ve vstupním řetězci se smažou resp. přečtou ze vstupu.
  3. **Přijetí:** konfigurace (e, e, π) znamená, že řetězec je rozpoznán, analýza končí a π obsahuje posloupnost pravidel reprezentující levou derivaci řetězce podle G.
  4. **Chyba:** ve všech ostatních případech analýza končí s chybovou signalizací.

  Pokusme se nyní provést SA slova vygenerovaného v gramatice z předchozího příkladu, ke které použijeme rozkladovou tabulku výše uvedenou.

Řešený příklad 9:

Mějme řetězec acbabbc, pak následující přechod konfigurací reprezentuje SA.

(acbabbc, S, e) ⇒ (acbabbc, aASc, 1) ⇒ (cbabbc, ASc, 1) ⇒
(cbabbc, cSAbSc, 14) ⇒ (babbc, SAbSc, 14) ⇒ (babbc, bAbSc, 142)
⇒ (abbc, AbSc, 142) ⇒ (abbc,abSc, 1423) ⇒ (bbc, bSc, 1423)
⇒ (bc, Sc, 1423) ⇒ (bc, bc, 14232) ⇒ (c, c, 14232) ⇒ (e, e, 14232)

3.3. Tvorba rozkladové tabulky

Vlastní algoritmus pro vytvoření rozkladové tabulky lze formalizovat následovně. Tento postup je ovšem použitelný pouze pro SLL(1) gramatiky.

Algoritmus 2: Vytvoření rozkladové tabulky pro SLL(1) gramatiku.

Vstup: SLL(1) gramatika G=(Π,Σ,S,P).

Výstup: rozkladová tabulka M pro G.

Vytvoření rozkladové tabulky

Postup:
- Rozkladová tabulka je definována na kartézském součinu $\Pi \times \Sigma$.
  1. Pokud $A \rightarrow a\alpha$ je i-té pravidlo v P, pak $M(A, a) = a\alpha$, i.

2. M(X, a) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Aplikujme nyní tyto postupy na trochu praktičtější problém. S pomocí velmi omezených prostředků, které nám dává SLL(1) gramatika se nám zápis i poměrně jednoduchých úloh bude konstruovat poměrně těžko. Nemáme totiž k dispozici klíčový prostředek, kterým je e-pravidlo (umožňuje provádět iteraci stejných výrazů) a zároveň nesmí dojít k situaci, kdy nějaké pravidlo začíná stejným terminálem. Tato kombinace činí z této úlohy v kontrastu s již řešenými podobnými gramatikami bez tohoto omezení poměrně složitý a méně přehledný problém.

Řešený příklad 10:

Sestrojme gramatiku, která bude popisovat blok v programovacím jazyce C, který bude složen ze sekvence abstraktních příkazů p nebo vnořených bloků (uzavřených do složených závorek – {,}). Aby bylo možné vůbec sestrojit SLL(1) gramatiku, musíme provést jisté omezení ukončení sekvence příkazů a bloků. V SLL(1) gramatice nemůžeme použít e-pravidlo a tedy je nutno odlišit situaci, kdy nějaký příkaz následuje a kdy ne. Upravíme si tedy deklaraci tak, že všechny příkazy jsou odděleny středníkem (;) vyjma posledního.

G = ({S, P, R},{p, {, }, ;}, S, P)
P: S $\rightarrow_1$ {P, P $\rightarrow_2$ pR, P $\rightarrow_3$ {PR, R $\rightarrow_4$;P, R $\rightarrow_5$ }

Konstrukce této gramatiky se opírá o následující pravidla:
- Neterminál S vytváří uzavření do závorek, ovšem musíme k ukončení použít jiných prostředků než u neomezených BKG, je zde totiž problém, jak rozlišit, že už jde o koncový příkaz nebo ještě blok pokračuje dalším.
- Sekvence příkazů je generována neterminálem P, rozlišují se dvě možnosti – buď jde o příkaz p nebo o vnořený blok začínající závorkou, oba konstrukty je možno ukončit pomocí R
- R vyžaduje rozlišení, zda ještě následuje další příkaz oddělený středníkem nebo jde o konec bloku uvozený uzavírací závorkou
- Jen díky tomuto poměrně nepřehlednému přístupu jsme dokázali sestrojit SLL(1) gramatiku, což navozuje myšlenku, že tyto gramatiky pro praktické úlohy jsou příliš omezené

Můžeme nyní odvodit ukázkový blok jazyka C:

S $\Rightarrow_1$ {P $\Rightarrow_2$ {pR $\Rightarrow_4$ {p;P $\Rightarrow_3$ {p;{PR$\Rightarrow_2$ {p;{pRR $\Rightarrow_5$ {p;{p}R
$\Rightarrow_4$ {p;{p};P} $\Rightarrow_4$ {p;{p};pR $\Rightarrow_5$ {p;{p};p}

Vidíte, že díky nelogickému rozdělení na část počáteční a koncovou je odvození složitější a nepřehlednější než u obecné BKG. Nyní proveďme kontrolu zda jde opravdu o SLL(1) gramatiku a pokud ano, tak sestrojíme rozkladovou tabulku a provedeme analýzu bloku, který jsme právě vygenerovali.

Kontrola podmínek pro SLL(1) gramatiku.

1. Všechna pravidla začínají terminálem.
2. U S je pouze jedno pravidlo a tudíž konflikt nehrozí. Neterminál P se přepisuje buď na řetězec začínající p nebo {, což opět není konflikt a neterminál R se přepisuje na řetězec začínající ; nebo }, což opět jsou různé symboly.

Gramatika tedy je SLL(1).

Rozkladovou tabulku sestrojíme dle algoritmu.

M	p	;	{	}
S			{P, 1	
P	pR, 2		{PR, 3	
R		;P, 4		}, 5

Nyní můžeme provést analýzu slova {p;{p};p}:

({p;{p};p}, S, e) ⇒ ({p;{p};p}, {P, 1) ⇒ (p;{p};p}, P, 1) ⇒
(p;{p};p}, pR, 12) ⇒ (;{p};p}, R, 12) ⇒ (;{p};p}, ;P , 124) ⇒
({p};p}, P, 124) ⇒ ({p};p}, {PR, 1243) ⇒ (p};p}, PR, 1243) ⇒
(p};p}, pRR, 12432) ⇒ (};p}, RR, 12432) ⇒ (};p}, }R, 124325) ⇒
(;p}, R, 124325) ⇒ (;p}, ;P, 1243254) ⇒ (p}, P, 1243254) ⇒
(p}, pR, 12432542) ⇒ (}, R, 12432542) ⇒ (}, }, 124325425)
⇒ (e, e, 124325425)

Slovo bylo rozpoznáno a levá derivace je reprezentována čísly použitých pravidel: 124325425

Výhodou SLL(1) gramatiky je samozřejmě velmi jednoduchá konstrukce rozkladové tabulky oproti složitějším gramatikám, které budeme probírat v následujících kapitolách. S pomocí této tabulky už je analýza zcela deterministická a mohl by ji velmi jednoduše provádět například počítačový program.

Nejdůležitější probrané pojmy:

- analýza „shora dolů“ a jednoznačnost
- LL gramatiky
- SLL(1) gramatika
- rozkladová tabulka
- algoritmus SA pro LL(1) gramatiky

Úkoly k textu:

1. Sestrojte SLL(1) gramatiku pro jazyk aritmetických výrazů s jediným operandem x s operací sčítaní a dále s možností vnořit místo operandu x podvýraz uzavřený do závorek o stejné struktuře.
2. K SLL(1) gramatice z úkolu 1. sestrojte rozkladovou tabulku a proveďte analýzu jednoduchého výrazu (s alespoň dvěmi spojkami a jedním vnořeným výrazem).

4. Q-gramatiky

Cíl:

Po prostudování této kapitoly pochopíte:
- co je q-gramatika
- jakou funkci a jaká omezení přináší e-pravidlo
- co je funkce FOLLOW a k čemu slouží

Naučíte se:
- vytvářet q-gramatiky pro problémové úlohy
- vytvářet rozkladové tabulky pro q-gramatiky
- vyčíslit funkci FOLLOW
- provádět SA pro q-gramatiky

V předcházející kapitole jsme se seznámili s nejjednodušším typem LL(1) gramatiky, který neumožňuje použití **epsilon pravidel**. Jak už bylo řečeno, jde o velmi omezující kritérium, neboť to kupříkladu neumožňuje zapsat žádnou gramatiku, ve které se vyskytuje prázdné slovo. To by ještě nebyl zásadní problém (lze se omezit i na takové gramatiky a samotné prázdné slovo řešit jinak – nesystémově). Bez epsilon pravidla nemůžeme přehledně popsat syntaktické struktury, které jsou obvyklé v problémových úlohách (viz předchozí kapitola). V následujícím textu si tedy ukážeme o něco obecnější gramatiky, které e-pravidlo připouštějí.

4.1. Q-gramatika a funkce FOLLOW

Takzvané q-gramatiky jsou vlastně SLL(1) gramatiky s rozšířením umožňujícím použít epsilon pravidlo. Toto použití ovšem není zcela automatické. Epsilon pravidlo může totiž způsobit ne zcela transparentní kolizi mezi tím, čím může pro určitý neterminál řetězec začínat a tím, co může následovat v generovaném slově v případě, že by se epsilon pravidlo aplikovalo (tudíž by se neterminál vymazal a následovat mohou **všechny řetězce**, které lze odvodit **bezprostředně za tímto neterminálem**).

To bude vyžadovat definici speciální funkce **FOLLOW**, která obsahuje všechny takové symboly, které následují. Zjištění, o které symboly jde, není triviální postup, avšak algoritmus lze zapsat několika pravidly.

Uvažujme následující gramatiku:

Řešený příklad 11:

G = ({S, A},{a, b, c}, S, P)

P: $S \rightarrow_1 aAS$, $S \rightarrow_2 b$, $A \rightarrow_3 cAS$, $A \rightarrow_4 e$

Tato gramatika není SLL(1), protože obsahuje epsilon pravidlo. Aby bylo možné provádět opět deterministickou SA podle stejného principu jako u SLL(1) gramatiky, musíme mít k dispozici rozkladovou tabulku. Vytvoření položek pro pravidla 1. – 3. se zdá velmi jednoduché a je totožné jako u SLL(1). Ale problematické je pravidlo 4. Kdy máme provést přepis podle něj a nezpůsobuje nám kolizi s jiným pravidlem pro neterminál A? Jak to poznáme?
Na tyto otázky existuje odpověď, pokud si uvědomíme, co bude znamenat aplikace tohoto pravidla. Pokud někde aplikujeme pravidlo 4., neterminál A v daném řetězci „zmizí“ a jako následující terminální symbol dostaneme množinu těch terminálů, které následují za A. Co do takovéto množiny patří. Uvažujme, kde se na pravé straně v pravidlech vyskytuje A. Jde o dva výskyty:

$S \rightarrow_1 aAS$ a $A \rightarrow_3 cAS$

V obou těchto případech vidíme, že pokud A „zmizí“ dostane se na jeho místo ten symbol, kterým „začíná“ neterminál S. U této gramatiky je pak zcela zřejmé, že S může díky pravidlům 1. a 2. začínat jedině symbolem a nebo b. Logicky se tedy tato pravidla aplikují, pokud se v generovaném/analyzovaném slově vyskytnou tyto symboly. Proto je zařadíme do příslušných sloupců pro neterminál A v rozkladové tabulce. Zároveň je jasné, že kdyby kterýkoliv z řetězců na pravé straně pravidla pro A začínal symbolem a nebo b, tak by nebyla tabulka jednoznačná, neboť se mohl provést buď přepis na tyto řetězce nebo na epsilon. Zde ovšem žádná nejednoznačnost nevzniká, neboť symboly a, b nekolidují s c. Tabulka ještě navíc musí obsahovat nový sloupec e (epsilon) a to z důvodu možnosti přepisu na zásobníku i v případě, že celé slovo už je přečteno a my můžeme ještě aplikovat epsilon pravidla (ty nevygenerují žádné symboly a tudíž nedojde k nesrovnalosti s obsahem zásobníku a analyzovaného řetězce).
Rozkladová tabulka tedy vypadá takto:

M	a	b	c	e
S	aAS, 1	b, 2		
A	e, 4	e, 4	cAS, 3	

Podle této tabulky můžeme analyzovat slovo aacbb:
(aacbb, S, e) $\Rightarrow$ (aacbb, aAS, 1) $\Rightarrow$ (acbb, AS, 1) $\Rightarrow$ (acbb, S, 14)
$\Rightarrow$ (acbb, aAS, 141)
$\Rightarrow$ (cbb, AS, 141) $\Rightarrow$ (cbb, cASS, 1413) $\Rightarrow$ (bb, ASS, 1413)
$\Rightarrow$ (bb, SS, 14134) $\Rightarrow$ (bb, bS, 141342) $\Rightarrow$ (b, S, 141342)
$\Rightarrow$ (b, b, 1413422) $\Rightarrow$ (e, e, 1413422)

Pozn. Každá aplikace pravidla 4 vyžadovala rozhodnutí, zda neterminál A vypustit. Je jasné, že kdyby například pro neterminál A a symbol b bylo definováno pravidlo, nedokázali bychom se rozhodnout zda použít ono konkrétní pravidlo nebo nejprve vypustit A a teprve následně b vygenerovat pomocí neterminálu nebo řetězce, který následuje! Podle symbolu, který se vykytuje jako následující v analyzovaném slově můžeme určit, které pravidlo by se použilo za předpokladu, že A se vypustí pomocí epsilon pravidla. V takovém případě, že tato možnost v kroku, který by následoval, existuje, můžeme díky epsilon pravidlu „uvolnit místo“ pro pozdější přepis na symbol následující ve slově. Proto se u q-gramatiky vyžaduje, aby množina FOLLOW pro neterminál, který přepisuje na epsilon, byla disjunktní s množinou symbolů, kterými začínají pravidla pro tento neterminál.

Definice 6: Máme neterminál X v G = (Π,Σ,P,S), pak platí:
FOLLOW(X) = {a| S⇒* αXβ, β ⇒*aγ, γ∈(Π ∪ Σ)* } ∪ {e| S⇒* αX}

Funkce FOLLOW

Hodnotou funkce FOLLOW pro daný neterminál X je množina všech symbolů, které mohu následovat za X, včetně e (epsilon), pokud za X už nemusí následovat žádný symbol (může být na konci řetězce v odvození).

Příkladem FOLLOW může být hodnota FOLLOW(A) = {a, b} pro gramatiku z předchozího příkladu.

Definice 7: BKG se nazývá q-gramatika, jestliže platí:

1. **Pravá strana pravidla je buď prázdná (epsilon) nebo začíná terminálem.**
2. **Každá dvě pravidla přepisující stejný neterminál X se liší terminálem, kterým začíná pravá strana (pokud X→aα, X→bβ jsou různá pravidla, pak a≠b).**
3. **Jestliže existuje pravidlo X→e, pak terminály, kterými začínají pravé strany ostatních pravidel X→aα, nesmí patřit do FOLLOW(X), a∉FOLLOW(X).**

Q-gramatika

4.2. Výpočet funkce FOLLOW

Výpočet funkce FOLLOW lze realizovat algoritmem, který pracuje podle následujících pravidel (jedná se o takzvaný tečkový algoritmus):

- tečka před symbolem v řetězci nám určuje místo, kde chceme určit terminální symbol, který následuje
- určíme si na začátku, které neterminály se mohou přepsat (libovolným počtem kroků) na e
- rozšiřujeme množinu pravidel s tečkou, tím, že přidáváme pravidla, která se vyskytují již v této množině a splňují daná kritéria

Výpočet FOLLOW

Algoritmus 3: Výpočet FOLLOW(A)
Vstup: BKG gramatika G=(Π,Σ,S,P) a neterminální symbol A
Výstup:FOLLOW(A)

Metoda:

1. Položíme N_e rovno množině všech prvků, které se dají přepsat (i rekurzivně) na prázdný řetězec (k tomu lze použít algoritmus z prvního dílu opory [Ha03], který se aplikoval při převodu BKG na nevypouštějící gramatiku).
2. Krok 2a: F = { A → A. }, do F umístíme fiktivní pravidlo, které reprezentuje situaci, kdy tečka vyjadřuje, že chceme zjistit všechny symboly, které se vyskytují za A.

   Krok 2b:

   Je-li v F pravidlo B → γ., kde γ je neprázdný řetězec, dáme do F všechna pravidla, ve kterých je na pravé straně B a tečku umístíme za B. V podstatě budeme dále určovat čím začíná řetězec za tečkou. Přidáváme tak vlastně všechny výskyty inkriminovaného neterminálu v celé gramatice, což je cílem.

   Krok 2c:

   Je-li v F pravidlo C → α.Bβ, přidáme do F všechna pravidla s B na levé straně, tečku umístíme na začátek. Tím, vlastně postupně rozvíjíme všechny neterminály na řetězce, na které se mohou přepsat.

   Krok 2d:

   Je-li v F pravidlo, kde je za tečkou neterminální symbol, který patří do N_e, do F vložíme toto pravidlo ještě jednou, ale tečku posuneme o jeden symbol doprava (to je případ, že tento neterminál se může přepsat na epsilon a tedy vypustit a je logické, že nás tedy zajímají i symboly bezprostředně za ním).

   Krok 2e:

   Kroky 2b, 2c, 2d opakujeme, dokud do F můžeme přidávat další položky.

   Krok 3:

   Do FOLLOW(A) dáme všechny terminální symboly, před kterými je tečka. Je-li v F pravidlo S → α., kde S je startovací symbol, přidáme do FOLLOW(A) i symbol e (epsilon) – tato situace

reprezentuje možnost, že za zkoumaným neterminálem už není žádný symbol, tj. S se přepisuje na řetězec, kterým zkoumané místo končí.

Aplikujme nyní algoritmus na gramatice z předchozího příkladu.

Řešený příklad 12:

G = ({S, A},{a, b, c}, S, P)
P: S $\rightarrow_1$ aAS, S $\rightarrow_2$ b, A $\rightarrow_3$ cAS, A $\rightarrow_4$ e

Určíme FOLLOW(A).

Krok 1: Spočítáme N_e = {A} – zjevně S se nemůže přepsat na e, neboť vždy obsahuje přepis na alespoň jeden terminál.

Krok 2a: F = { A $\rightarrow$ A. }
Krok 2b: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S }
Krok 2c: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S, S $\rightarrow_1$ .aAS, S $\rightarrow_2$ .b }
Krok 2d: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S, S $\rightarrow_1$ .aAS, S $\rightarrow_2$ .b } – nelze nic přidat, neboť tečka není nikde před A.
Krok 2e: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S, S $\rightarrow_1$ .aAS, S $\rightarrow_2$ .b } – konec, protože už není možno v následujícím kroku přidat žádnou novou položku kroky 2b.,2c. ani 2d.
Krok 3: FOLLOW(A) = {a,b}

Výpočet funkce FOLLOW použijeme ve dvou případech:

1. Je nutný pro zjištění, zda gramatika je q-gramatika (viz podmínka 3 definice).
2. Umožní do rozkladové tabulky vložit buňky, odpovídající epsilon pravidlu pro daný neterminál.

4.3. Tvorba rozkladové tabulky

Vytvoření rozkladové tabulky pro q-gramatiku je složitější než pro SLL(1). Složitost spočívá v nutnosti správně zaplnit buňky odpovídající přepisu podle pravidel s epsilon, což vyžaduje spočítat funkce FOLLOW pro všechny neterminály, které přepisují na e. Pak toto pravidlo vložíme do sloupců odpovídajících symbolům v množině FOLLOW, resp. epsilon, pokud FOLLOW e obsahuje.

Algoritmus 4: Vytvoření rozkladové tabulky pro q-gramatiku.

Vstup: q-gramatika G=(Π,Σ,S,P).

Výstup: rozkladová tabulka M pro G.

Rozkladová tabulka pro Q-gramatiku

Postup:
- Rozkladová tabulka je definována na kartézském součinu $\Pi \times (\Sigma \cup e)$.
  1. Pokud $A \rightarrow a\alpha$ je i-té pravidlo v P, pak $M(A, a) = a\alpha, i$.
  2. Pokud $A \rightarrow e$ je i-té pravidlo v P, pak $M(A, b) = e, i$ pro všechny $b \in FOLLOW(A)$.
  3. $M(X, a)$ = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Aplikujme nyní probrané postupy na příkladu analogickém jako v předchozí kapitole (s mírnou modifikací).

Řešený příklad 13:

Sestrojme gramatiku, která bude popisovat blok v programovacím jazyce C, který bude složen ze sekvence abstraktních příkazů p nebo vnořených bloků (uzavřených do složených závorek – {,}). Na rozdíl od konstruované q-gramatiky se již nemusíme omezovat v deklaraci jazyka. Můžeme dovolit použití epsilon a tudíž ukončení lze realizovat bez explicitního neterminálu a navíc není třeba vyžadovat, aby poslední příkaz bloku neobsahoval ;. Také je možné, aby příkaz byl prázdný – tedy pouze středník, resp. blok může být prázdný.

$G = (\{S, P, R\}, \{p, \{, \}, ;\}, S, P)$
P: $S \rightarrow_1 \{P\}$, $P \rightarrow_2 p;P$, $P \rightarrow_3 \{P\}P$, $P \rightarrow_4 ;P$, $P \rightarrow_5 e$

Pozn. Neterminál P vyjadřuje všechny možnosti, jak může vypadat sekvence příkazů.

Můžeme nyní odvodit ukázkový blok jazyka C:

$S \Rightarrow_1 \{P\} \Rightarrow_2 \{p;P\} \Rightarrow_3 \{p;\{P\}P\} \Rightarrow_2 \{p;\{p;P\}P\} \Rightarrow_5 \{p;\{p;\}P\}$
$\Rightarrow_4 \{p;\{p\};P\} \Rightarrow_2 \{p;\{p\};p;P\} \Rightarrow_5 \{p;\{p\};p;\}$

Tato gramatika je schopna mnohem přehledněji generovat bloky a sekvence příkazů a navíc lépe odpovídá normě jazyka C.

Kontrola podmínek pro q-gramatiku. Potřebujeme vyčíslit FOLLOW(P):

Krok 1: Spočítáme $N_e = \{P\}$ – zjevně S se nemůže přepsat na e, neboť vždy obsahuje přepis na alespoň jeden terminál.

Krok 2a: $F = \{ P \rightarrow P. \}$
Krok 2b: $F = \{ P \rightarrow P. , S \rightarrow_1 \{P.\}, P \rightarrow_2 p;P. , P \rightarrow_3 \{P.\}P, P \rightarrow_3 \{P\}P.,$
$P \rightarrow_4 ;P. \}$
Krok 2c: nic nového se nepřidá
Krok 2d: nic nového se nepřidá
Krok 2e: : $F = \{ P \rightarrow P. , S \rightarrow_1 \{P.\}, P \rightarrow_2 p;P. , P \rightarrow_3 \{P.\}P, P \rightarrow_3 \{P\}P.,$

$P \rightarrow_4$;P. }
– konec, protože už není možno v následujícím kroku přidat žádnou novou položku kroky 2b.,2c. ani 2d.
Krok 3: FOLLOW(A) = { '}' } – obsahuje pouze symbol }

1. Všechna pravidla začínají terminálem nebo epsilonem.
2. U S je pouze jedno pravidlo a tudíž konflikt nehrozí. Neterminál P se přepisuje buď na řetězec začínající p, { nebo ; , což opět není konflikt.
3. Nesmí být konflikt mezi množinou FOLLOW(P) a všemi terminály, kterými začínají pravidla 2. – 4. Konflikt není, protože ve FOLLOW(P) je jen symbol } a ten u žádného z těchto pravidel na začátku není.

Gramatika tedy je q-gramatika.

Rozkladovou tabulku sestrojíme dle algoritmu.

M	p	;	{	}	e
S			{P}, 1		
P	p;P, 2	;P, 4	{P}P, 3	e, 5	

Nyní můžeme provést analýzu slova {p;{p;};p;}:

({p;{p;};p;}, S, e) ⇒ ({p;{p;};p;}, {P}, 1) ⇒ (p;{p;};p;}, P}, 1)
⇒ (p;{p;};p;}, p;P}, 12) ⇒ (;{p;};p;}, ;P}, 12) ⇒ ({p;};p;}, P}, 12)
⇒ ({p;};p;}, {P}P}, 123) ⇒ (p;};p;}, P}P}, 123)
⇒ (p;};p;}, p;P}P}, 1232) ⇒ (;};p;}, ;P}P}, 1232)
⇒ (};p;}, P}P}, 1232) ⇒ (};p;}, }P}, 12325) ⇒ (;p;}, P}, 12325)
⇒ (;p;}, P}, 12325) ⇒ (;p;}, ;P}, 123254) ⇒ (p;}, P}, 123254)
⇒ (p;}, p;P}, 1232542) ⇒ (;}, ;P}, 1232542) ⇒ (}, P}, 1232542)
⇒ (}, }, 12325425) ⇒ (e, e, 12325425)

Slovo bylo rozpoznáno a levá derivace je reprezentována čísly použitých pravidel: 112325425; každá z aplikací pravidla 5. vyjadřuje situaci, kdy došlo k umazání/ukončení P na konci bloku.

Nejdůležitější probrané pojmy:
- q-gramatika a její rozkladová tabulka
- Funkce FOLLOW

Úkoly k textu:

1. Sestrojte q-gramatiku pro jazyk aritmetických výrazů s jediným operandem x s operací sčítaní, násobení a dále s možností vnořit místo operandu x podvýraz uzavřený do závorek o stejné struktuře.
2. Ke q-gramatice z úkolu 1. sestrojte rozkladovou tabulku a proveďte analýzu jednoduchého výrazu (s alespoň dvěmi spojkami a jedním vnořeným výrazem).

5. LL(1) gramatiky

Cíl:

Po prostudování této kapitoly pochopíte:
- co je LL(1) gramatika
- jakou funkci a jaká omezení přináší neterminál na začátku pravidla
- co je funkce FIRST a k čemu slouží

Naučíte se:
- vytvářet LL(1) gramatiky pro problémové úlohy
- vytvářet rozkladové tabulky pro LL(1) gramatiky
- vyčíslit funkci FIRST
- provádět SA pro LL(1) gramatiky

Q-gramatiky mají již poměrně vysokou expresivitu, jak jsme viděli na problémových úlohách. Mají však ještě jednu nevýhodu, která se plně projevuje až u složitějších (rozsáhlejších gramatik). Touto nevýhodou je nutnost, aby každé pravidlo začínalo terminálem, což může vést také k jisté nepřehlednosti a nesystematičnosti gramatiky. Již dopředu (před studiem obecných vlastností LL gramatik) lze ale říci, že tento požadavek je už opravdu spíše otázkou „komfortnosti“ návrhu a zápisu gramatiku, protože libovolnou LL(1) gramatiku lze jednoduchým algoritmem převést na q-gramatiku (na rozdíl od vztahu q-gramatik a SLL(1) gramatik).

5.1. Funkce FIRST

LL(1) gramatika umožňuje, aby pravidlo začínalo neterminálem. Přesto stále musíme trvat na požadavku, aby i mezi takovými pravidly pro jeden neterminál nevznikala kolize. Jak ale takovou situaci odhalit a následně tvořit rozkladovou tabulku? Je k tomu potřeba opět jistá funkce, která se nazývá **FIRST**. Vyjadřuje množinu pro daný řetězec, která obsahuje všechny terminální symboly resp. epsilon, kterými může řetězec začínat. Způsob jejího formálního výpočtu je podobný jako u FOLLOW – postupně procházíme pravidla a přidáváme na základě jistých pravidel do množiny položky s tečkou a na konci tyto položky projdeme a zjistíme terminály, které jsou bezprostředně za tečkou.
Uvažujme následující příklad.

Řešený příklad 14:

Mějme gramatiku pro tvorbu aritmetických výrazů s operandem x, operací sčítání a vnořenými podvýrazy se závorkami.

G = ({S, A, B},{x, +, (,)}, S, P)

P: S $\rightarrow_1$ BA, A $\rightarrow_2$ + BA, A $\rightarrow_3$ e, B $\rightarrow_4$ x, B $\rightarrow_5$ (S)

Pozn. V této gramatice B reprezentuje operandy a umožňuje generovat libovolně mnoho operandů spojených symbolem +.

Tato gramatika není zjevně ani SLL(1) ani q-gramatika, neboť pravidlo 1. nezačíná terminálem. Přesto pro ni lze poměrně analogicky sestrojit rozkladovou tabulku, pokud se nám podaří zjistit čím začíná řetězec BA. řetězec začíná na B a tedy pohledem na pravidla pro B zjišťujeme, že B může začínat jedinými dvěma terminály – x a (. Sestrojení rozkladové tabulky by se pak ubíralo stejnými pravidly, jako u q-gramatiky. Musíme vyčíslit FOLLOW(A), protože A se přepisuje na epsilon. FOLLOW(A)={), e}, protože za A následuje uzavírací závorka a navíc se může A vyskytnout zcela na konci slova (viz pravidlo 1.).

Rozkladová tabulka by pak vypadalo takto:

M	x	+	(	)	e
S	BA, 1		BA, 1		
A		+ BA, 2		e, 3	e, 3
B	x, 4		(S), 5		

Funkce FIRST

Definice 8: Máme řetězec $\alpha \in (\Pi \cup \Sigma)^*$ v G = (Π,Σ,P,S), pak platí: FIRST(α) = {a| $\alpha \Rightarrow^* a\beta$, $a \in \Sigma$, $\beta \in (\Pi \cup \Sigma)^*$ } ∪ {e| $\alpha \Rightarrow^*$ e}

Funkce FIRST pro daný řetězec je daná množinou terminálů, kterými může řetězec po odvození začínat, resp. epsilon, pokud se může derivovat na prázdný řetězec.

Výpočet FIRST lze realizovat opět jednoduchým algoritmem, který vychází z „tečkové“ notace, kde tečka vyjadřuje místo, které chceme prozkoumat.

Výpočet funkce FIRST

Algoritmus 5: Výpočet funkce FIRST
Vstup: BKG gramatika G=(Π,Σ,S,P) a řetězec $\alpha = X_1X_2\ldots X_n$
Výstup:FIRST(α)

Metoda:

Krok 1a: Položíme množinu F={. $X_1X_2\ldots X_n$ }

Krok 1b:

Je-li v F pravidlo $A \rightarrow \beta.A\gamma$, kde $A \in \Pi$, přidáme do F všechna pravidla $A \rightarrow .\delta$, kde $\delta \in (\Pi \cup \Sigma)^*$

Tento krok reprezentuje výpis všech možností přepisu neterminálu, kterým může začínat řetězec – je před ním tečka, čímž získáme nové položky pro prozkoumání.

Krok 1c:

Je-li v F pravidlo $B \rightarrow \delta.$, kde $\delta \in (\Pi \cup \Sigma)^*$, přidáme do F pravidla z původní množiny F, ve kterých se vyskytoval symbol B, ale tečku umístíme až za něj. To se stane ve dvou případech:

- $\delta = e$ – a to tedy tečka můžeme pod neterminálem „podplavat“, neboť neterminál může být vypuštěn,
- $\delta \neq e$ – to stane pokud se všechny neterminály v řetězci mohly vypustit a tím pádem nastává stejný efekt jako v prvním případě.

Krok 1d:

Kroky 1b a 1c se opakují tak dlouho, dokud lze do F přidávat nové položky.

Krok 2:

FIRST(α) bude obsahovat všechny terminální symboly z F, které jsou bezprostředně za tečkou. Zároveň přidáme e, je-li tečka na konci pravidla.

Aplikujme tento algoritmus na předchozí gramatiku.

Řešený příklad 15:

G = ({S, A, B},{x, +, (,)}, S, P)
P: $S \rightarrow_1 BA$, $A \rightarrow_2 + BA$, $A \rightarrow_3 e$, $B \rightarrow_4 x$, $B \rightarrow_5 (S)$

Vypočtěme FIRST(BA).

Krok 1a: F={.BA}
Krok 1b: F={.BA, $B \rightarrow_4 .x$, $B \rightarrow_5 .(S)$ }
Krok 1c: nelze nic nového přidat.
Krok 1d: v dalším kroku by nic nového nebylo přidáno.
Krok 2: FIRST(BA)={x, (}

5.2. LL(1) gramatika

I LL(1) gramatika musí splňovat pravidlo, aby v její rozkladové tabulce nedošlo k žádné kolizi. Lze to formulovat slovně tak, že:

1. Pokud existují dvě pravidla pro jeden neterminál, řetězce na pravé straně musí začínat různými terminálními symboly (nemusí jít o explicitně zapsané terminální symboly!) Říkáme, že nesmí nastat tzv. **FIRST-FIRST kolize**.

FIRST-FIRST kolize

FIRST-FOLLOW kolize

2. Pokud se nějaký neterminál přepisuje na epsilon, pak všechna pravidla pro tento neterminál musí začínat jiným symbolem než který za neterminálem může následovat. (podmínka analogická jako u q-gramatiky). Nesmí nastat tzv. **FIRST-FOLLOW kolize**.

Přesně to lze elegantně formulovat pomocí FIRST a FOLLOW v definici.

LL(1) gramatika

Definice 9: BKG G=(Π,Σ,S,P) se nazývá LL(1) gramatika, jestliže platí pro každé X ∈ Π, kde v P jsou různá pravidla X→ α, X→ β :

1. **FIRST(α) ∩ FIRST(β) = ∅.**
2. **Pokud z řetězce α je možné generovat prázdný řetězec a z řetězce β není možné generovat prázdný řetězec, pak FOLLOW(A) ∩ FIRST(β) = ∅.**

Pozn. Obě podmínky lze ještě integrovat do sebe tak, že se vyjádří společnou ekvivalentní podmínkou:
FIRST(αFOLLOW(A)) ∩ FIRST(βFOLLOW(A)) = ∅.

5.3. Tvorba rozkladové tabulky

Pokud máme vytvořeny množiny FIRST všech řetězců, které se vyskytují na pravé straně pravidel, a množiny FOLLOW pro neterminály, které se přepisují na epsilon, je poměrně snadné vytvořit rozkladovou tabulku. Zjištění množin FIRST pro řetězce, které začínají terminálem je triviální a tudíž není nutné vždy aplikovat důsledně algoritmus na výpočet FIRST. U složitějších gramatik a řetězců, kde je na začátku neterminál je rozhdně bezpečnější provést výpočet dle algoritmu než odhadnou množinu pouhým pohledem na přepisovací pravidla. Ještě více to platí o FOLLOW, jejíž výpočet je o něco složitější.
Tvorba rozkladové tabulky se opírá o následující dvě pravidla:

1. Pro pravidla, která přepisují na neprázdné řetězce spočítáme FIRST těchto řetězců a do sloupců příslušných sloupců toto pravidlo vložíme. (výjimkou je sloupec epsilon, kam nepřidáváme nic – to je situace která se může vyskytnout na konci řetězce a tu řeší epsilon pravidla)
2. Pro pravidla, která přepisují na prázdné řetězce spočítáme FOLLOW neterminálu, který přepisují a vložíme toto pravidlo do příslušných sloupců symbolů resp. epsilonu pro nalezené prvky FOLLOW.

Algoritmus 6: Vytvoření rozkladové tabulky pro LL(1) gramatiku.

Rozkladová tabulka pro LL(1) gramatiku

Vstup: LL(1) gramatika G=(Π,Σ,S,P).

Výstup: rozkladová tabulka M pro G.

Postup:

- Rozkladová tabulka je definována na kartézském součinu $\Pi \times (\Sigma \cup e)$.
  1. Pokud $A \rightarrow \alpha$ je i-té pravidlo v P, pak $M(A, a) = \alpha$, i pro všechny $a \in FIRST(\alpha) - \{e\}$.
  2. Pokud $A \rightarrow \alpha$ je i-té pravidlo v P a $e \in FIRST(\alpha)$, pak $M(A, b) = \alpha$,i pro všechny $b \in FOLLOW(A)$.
  3. M(X, a) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Aplikujme nyní všechny postupy na problémové úloze.

Řešený příklad 16:

Sestrojme gramatiku pro generování aritmetických výrazů s operacemi sčítaní, násobení a operandem x, umožňující navíc vnořovat podvýrazy stejného typu pomocí závorek.

G = ({S, A, B},{x, *, +, (,)}, S, P)
P: $S \rightarrow_1 AP$,
$P \rightarrow_2 + AP$, $P \rightarrow_3 e$
$A \rightarrow_4 BR$,
$R \rightarrow_5 * BR$, $R \rightarrow_6 e$,
$B \rightarrow_7 (S)$, $B \rightarrow_8 x$

V této gramatice se na rozdíl od Řešený příklad 1: musí využít jiný způsob na vytváření opakovaného generování sčítaní a násobení. Nelze použít rekurzivní volání přímo S resp. A, protože by tím vznikla kolize u dvou pravidel, které by obě začínaly stejným řetězcem A resp. B. Proto se musí zavést nové neterminály P resp. R, které vlastně umožňují hrát roli jakéhosi vytýkaní neterimálu S resp. A, čímž odstraníme kolizi, tím, že se zbavíme dvou pravidel u S resp. A. U nově vzniklých neterminálů P resp. R kolize FIRST-FIRST nastat nemůže, neboť jedno z pravidel přepisuje na epsilon. Může však potenciálně nastat FIRST-FOLLOW kolize, díky ukončovacímu pravidlu s e. Jak ale ihned ukážeme, nedochází k ní ani v jednom případě.

Kontrola podmínek pro LL(1) gramatiku a vytvoření FIRST a FOLLOW množin, které budeme potřebovat i pro konstrukci rozkladové tabulky:

Nejprve určíme triviální množiny FIRST:
P2. FIRST(+AP) = {+}, P5. FIRST(*BR) = {*}, P7. FIRST((S)) = {(}, P8. FIRST(x) = {x}
Dále pomocí algoritmů určíme netriviální množiny FIRST:
P1. FIRST(AP)
Krok 1a: F={.AP}, krok 1b: F={.AP, $A \rightarrow_4 .BR$ }, krok 1c.: nic, krok 1d: opakujeme 1b. F={.AP, $A \rightarrow_4 .BR$, $B \rightarrow_7 .(S)$, $B \rightarrow_8 .x$ }, krok 1c.: nic, krok 1d.: už nic nového nemůžeme následným krokem přidat.

FIRST(AP)={(, x}

P4. FIRST(BR)
Krok 1a: F={.BR}, krok 1b: F={.BR, B $\rightarrow_7$.(S), B $\rightarrow_8$ .x }, krok 1c.: nic, krok 1d: už nic nového nemůžeme následným krokem přidat.
FIRST(BR)={(, x}

A nakonec zbývá nejsložitější výpočet FOLLOW.

P3. FOLLOW(P)
Krok 1: N_e={P, R}
Krok 2a: F={P $\rightarrow$ P.},
Krok 2b: F={P $\rightarrow$ P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP.} – přidaly se pravidla, kde se vyskytuje P,
Krok 2c: F={P $\rightarrow$ P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP.} – nic se nepřidá
Krok 2d: F={P $\rightarrow$ P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP.} – nic se nepřidá
Krok 2e: opakujeme od 2b.
Krok 2b: F={P $\rightarrow$ P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP., B $\rightarrow_7$ (S.)} – přidaly se pravidla, kde se vyskytuje S,
Krok 2c, 2d: F={P $\rightarrow$ P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP., B $\rightarrow_7$ (S.)} – nic se nepřidá
Krok 2e: v následujícím průchodu už se nic nepřidá.

FOLLOW(P)={), e} – epsilon patří do množiny, neboť se vyskytla položka, kde se S přepisuje na řetězec s tečkou na konci

P6. FOLLOW(R)
Krok 1: N_e={P, R}
Krok 2a: F={R $\rightarrow$ R.},
Krok 2b: F={R $\rightarrow$ R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR.,} – přidaly se pravidla, kde se vyskytuje R,
Krok 2c: F={R $\rightarrow$ R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR.} – nic se nepřidá
Krok 2d: F={R $\rightarrow$ R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR.} – nic se nepřidá
Krok 2e: opakujeme od 2b.
Krok 2b: F={R $\rightarrow$ R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P }– přidaly se pravidla, kde se vyskytuje A,
Krok 2c: F={R $\rightarrow$ R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P,
P $\rightarrow_2$.+ AP, P $\rightarrow_3$ .e }– přidaly se pravidla,která přepisují P,
Krok 2d: F={R $\rightarrow$ R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P,
P $\rightarrow_2$.+ AP, P $\rightarrow_3$ .e, S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP. }– přidaly se pravidla, kde se tečka přesunula přes P, neboť je v N_e,
Krok 2e: opakujeme od 2b.
Krok 2b: F={R $\rightarrow$ R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P,
P $\rightarrow_2$.+ AP, P $\rightarrow_3$ .e, S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP., B $\rightarrow_7$ (S.) } – vyskytla se nová položka s tečkou na konci pro S a P
Kroky 2c. a 2d.: nic nového se nepřidá

Krok 2e: v následujícím průchodu už se nic nepřidá.

FOLLOW(R)={+,), e} – epsilon patří do množiny, neboť se vyskytla položka, kde se S přepisuje na řetězec s tečkou na konci

Nyní zkontroluje podmínky pro LL(1).

1. Jediný neterminál, kterého hrozí FIRST-FIRST kolize je B. Ovšem pro pravidlo 7. a 8. FIRST((S)) = {(} a FIRST(x) = {x}, a tedy podmínka je splněna.
2. FIRST-FOLLOW kolize hrozí jednak u P a R.

FIRST(+AP) = {+} a FOLLOW(P)={), e} – kolize nenastala.
FIRST(*BR) = {*} a FOLLOW(R)={+,), e} – kolize nenastala.

Gramatika tedy je LL(1).

Nyní sestrojíme rozkladovou tabulku.

M	x	+	*	(	)	e
S	AP, 1			AP, 1		
P		+ AP, 2			e, 3	e, 3
A	BR, 4			BR, 4		
B	x, 8			(S), 7		
R		e, 6	*BR, 5		e, 6	e, 6

Analyzujeme ukázkový výraz (x + x) * x.
((x + x) * x , S, e) ⇒ ((x + x) * x , AP, 1) ⇒ ((x + x) * x , BRP, 14) ⇒
((x + x) * x , (S)RP, 147) ⇒ (x + x) * x , S)RP, 147) ⇒
(x + x) * x , AP)RP, 1471) ⇒ (x + x) * x , BRP)RP, 14714) ⇒
(x + x) * x , xRP)RP, 147148) ⇒ (+ x) * x , RP)RP, 147148) ⇒
(+ x) * x , P)RP, 1471486) ⇒ (+ x) * x , +AP)RP, 14714862) ⇒
(x) * x , AP)RP, 14714862) ⇒ (x) * x , BRP)RP, 147148624) ⇒
(x) * x , xRP)RP, 1471486248) ⇒ ( ) * x , RP)RP, 1471486248) ⇒
( ) * x , P)RP, 14714862486) ⇒ ( ) * x ,)RP, 147148624863) ⇒
(* x , RP, 147148624863) ⇒ (* x , *BRP, 147148624863) ⇒
(x , BRP, 147148624863) ⇒ (x , xRP, 1471486248638) ⇒
(e , RP, 1471486248638) ⇒ (e , P, 14714862486386) ⇒
(e , e, 147148624863863)

Slovo bylo rozpoznáno a výsledná sekvence 147148624863863 reprezentuje levé odvození.

Nejdůležitější probrané pojmy:

- LL(1) gramatika a její rozkladová tabulka

- Funkce FIRST
- FIRST-FIRST kolize
- FIRST-FOLLOW kolize

Korespondenční úkol:

Sestrojte bezkontextovou gramatiku a Backusovu-Naurovou formu pro jednoduchý programovací jazyk složený ze seznamu řádků s příkazy. Řádek je uvozen návěštím ve tvaru X: příkaz, kde X je přirozené číslo. Lze používat identifikátory, které začínají písmenem anglické abecedy a obsahují libovolně mnoho písmen a číslic. Příkazy které se mohou použít jsou následující:

- přiřazovací příkaz tvaru X = aritmetický výraz, kde X je identifikátor a aritmetický výraz je výraz obsahující operandy – identifikátory a dále přirozená čísla, operace sčítání (+), odčítání (-), násobení (*) a dělení (/) a také umožňují vnořovat podvýrazy pomocí závorek.
- Nepodmíněný skok tvaru > X, kde X je návěští řádku, na který se má skočit.
- Podmíněný skok tvaru ?X$Y, kde X je identifikátor a Y je návěští a význam tohoto příkazu je, že se skočí na Y pouze pokud hodnota identifikátoru X je nula.
- Příkaz ukončení běhu programu - !

Pozn. I když pro řešení tohoto to nepotřebujeme vědět, protože používáme pouze syntaxi jazyka, všechny operace „ořezávají“ výsledné číslo na nezáporné hodnoty.

Pro Vámi sestrojenou LL(1) gramatiku proveďte kontrolu, zda je LL(1) a následně sestrojte rozkladovou tabulku a analyzujte jednoduchý ukázkový program:

```
10:X=5
20:Y=1
30:Y=X*Y
40:X=X–1
50 ?X$70
60:>30
70:!
```

6. Silné a slabé LL(k) gramatiky

Cíl:

Po prostudování této kapitoly pochopíte:
- Pojem obecné LL(k) gramatiky
- Rozdíl mezi silnou a slabou LL(k) gramatikou
- Pojem funkce $FIRST_k$ a $FOLLOW_k$

Naučíte se:

- Vytvářet silné LL(k) gramatiky
- Vytvářet rozkladové tabulky a provádět SA pro silné LL(k)
- Způsob SA pro slabé LL(k) gramatiky

V předchozích kapitolách jsme se zabývali gramatikami, pro které lze poměrně jednoduše provádět SA pouze s informací, jaký následuje v analyzovaném slově první symbol. To je samozřejmě velice pohodlné a jak uvidíme dají se do tohoto tvaru převést gramatiky pro praktické problémy (programovací jazyky, výrazy atd.). Přesto existují i LL gramatiky, které nelze analyzovat s informací o jediném symbolu, ale s informací o k-symbolech následujících ve slově (říká se jim silné LL(k) gramatiky). A dokonce existují pro každé takové k i gramatiky, kde nám nestačí znát pouze informaci o následujících k-symbolech, ale musíme znát a provádět rozhodnutí na základě dosavadního průběhu samotné analýzy.

6.1. Funkce $FIRST_k$ a $FOLLOW_k$

Abychom mohli rozšíření na k-symbolů následujících ve slově provést, musíme pochopitelně rozšířit funkce FIRST a FOLLOW.

$FIRST_k$
$FOLLOW_k$

Definice 10: Máme řetězec $\alpha \in (\Pi \cup \Sigma)^*$ a neterminál A v G = (Π,Σ,P,S), pak platí:

$$\mathbf{FIRST_k(\alpha) = \{x|\ \alpha \Rightarrow^* x\beta,\ x \in \Sigma^*,\ |x| = k,\ \beta \in (\Pi \cup \Sigma)^*\ \} \cup \{x|\ \alpha \Rightarrow^* x,\ x \in \Sigma^*,\ |x| < k\}}$$

$$\mathbf{FOLLOW_k(A) = \{x|\ S \Rightarrow^* wAx\beta,\ x \in \Sigma^*,\ |x| = k,\ w,\beta \in (\Pi \cup \Sigma)^*\ \} \cup \{x|\ S \Rightarrow^* wAx,\ x \in \Sigma^*,\ |x| < k,\ w,\beta \in (\Pi \cup \Sigma)^*\ \}}$$

Množina pro funkci $\mathbf{FIRST_k(\alpha)}$ tedy obsahuje jednak všechny řetězce o velikosti k, které se mohou vyskytovat na začátku řetězce α a jednak všechny řetězce o velikosti menší než k, pokud se takový řetězec vyskytuje na konci slova (už za ním nic nenásleduje). Může tedy jít o poměrně

velkou a náročně se hledající množinu všech kombinací symbolů splňujících tyto dvě podmínky.
Množina pro funkci **$FOLLOW_k(A)$** obsahuje jednak všechny řetězce o velikosti k, které se mohou vyskytovat bezprostředně za A a jednak všechny řetězce o velikosti menší než k, pokud se takový řetězec vyskytuje na konci slova.

Algoritmy pro výpočet těchto funkcí nebudeme explicitně uvádět. Postačí slovní vyjádření pomocí modifikací algoritmů pro FIRST a FOLLOW. Jelikož chceme získat nejen symboly na první pozici, ale potenciálně na k pozicích, obohatíme algoritmy o další opakující se krok.

Modifikace algoritmů FIRST a FOLLOW pro výpočet $FIRST_k$ a $FOLLOW_k$.

V algoritmech se místo obyčejné tečky, bude používat teček s indexy. Všechny tečky bez indexu se považují za tečku s indexem 1. Provádíme-li operace přidání položky do F, index tečky se kopíruje.
Do algoritmů pro výpočet FIRST a FOLLOW přidáme následující kroky:

- Pokud se v položce v množině F vyskytuje tečka s indexem i před terminálním symbolem, umístíme další tečku s indexem i+1 za tento terminální symbol.
- Nevytváříme nikdy položky s tečkou s indexem vyšším než k.

Výsledná množina se modifikuje tak, že do ní budou patřit všechna terminální slova o délce maximálně k vyskytující se za tečkou s indexem 1 s vyloučením všech teček z těchto slov.

Dále definujeme funkce $FIRST_k$ a $FOLLOW_k$ pro celé množiny řetězců resp. symbolů.

Definice 11: Máme podmnožinu řetězců $R \subseteq (\Pi \cup \Sigma)^*$ v $G=(\Pi,\Sigma,P,S)$, pak platí:
$FIRST_k(R) = \{x \mid x \in FIRST_k(\alpha)$ pro nějaké $\alpha \in R\}$
$FOLLOW_k(R) = \{x \mid x \in FOLLOW_k(\alpha)$ pro nějaké $\alpha \in R\}$

6.2. Silné LL(k) gramatiky a jejich SA

Silná LL(k) gramatika musí splňovat podobná kritéria jako LL(1) gramatika, avšak rozšířená na k-symbolů.

Silná LL(k) gramatika

Definice 12: BKG $G=(\Pi,\Sigma,S,P)$ se nazývá silná LL(k) gramatika, jestliže platí pro každé $X \in \Pi$, kde v P jsou různá pravidla $X \rightarrow \alpha$, $X \rightarrow \beta$:
$FIRST_k(\alpha FOLLOW_k(A)) \cap FIRST_k(\beta FOLLOW_k(A)) = \varnothing$.

Pro tyto gramatiky pak můžeme i použít velmi podobný algoritmus na vytvoření rozkladové tabulky (hlavní rozdíl spočívá ve struktuře, kde mohou být nejen jednotlivé symboly ve sloupcích, ale celé řetězce). Pro jednodušší definici zavedeme tedy množinu všech řetězců s omezenou délkou $\Sigma^{*k} = \{x \mid x \in \Sigma^*, |x| <= k\}$.

Rozkladová tabulka pro silnou LL(k) gramatiku

Algoritmus 7: Vytvoření rozkladové tabulky pro silnou LL(k) gramatiku.

Vstup: LL(k) gramatika G=(Π,Σ,S,P).

Výstup: rozkladová tabulka M pro G definovaná na $\Pi \times \Sigma^{*k}$.

Postup:

- Rozkladová tabulka je definována na kartézském součinu $\Pi \times (\Sigma \cup e)$.
  1. Pokud $A \rightarrow \alpha$ je i-té pravidlo v P, pak $M(A, x) = \alpha$, i pro všechny $x \in FIRST_k(\alpha)$, kde $|x| = k$.
  2. Pokud $A \rightarrow \alpha$ je i-té pravidlo v P a $y \in FIRST_k(\alpha)$ a $|y| < k$, pak $M(A, z) = \alpha$,i pro všechny $z \in FIRST_k(\alpha FOLLOW_k(A))$.
  3. M(X, y) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

První podmínka tedy definuje všechny přechody pro řetězce o velikosti k. V druhé pak doplňujeme přechody pro řetězce o délce menší než k – samozřejmě včetně epsilon pravidel.

Abychom mohli provádět syntaktickou analýzu, je potřeba ještě nadefinovat modifikovaný algoritmus SA.

SA pro silnou LL(k) gramatiku

Algoritmus 8: Syntaktická analýza pro silné LL(k) gramatiky

Vstup: rozkladová tabulka M pro silnou LL(k) gramatiku G=(Π,Σ,S,P), vstupní řetězec $w \in \Sigma^*$.

Výstup: levý rozklad (derivace) vstupního řetězce v případě, že $w \in L(G)$, jinak chybová signalizace.

Postup:

- Algoritmus čte vstupní řetězec, používá zásobník a má k dispozici slovo $u = FIRST_k(x)$, kde x je dosud nepřečtená část řetězce.
- Počáteční situace je (w, S, e).
- Vykonávají se přechody podle 1. a 2. dokud nenastane situace 3. nebo 4.
  1. **Expanze:** $(x, A\alpha, \pi) \Rightarrow (x, \beta\alpha, \pi i)$, pokud $A \in \Pi$, $M(A, u) = \beta$, i. Symbol A se na vrcholu zásobníku nahradí řetězcem β a číslo i je připojeno k posloupnosti reprezentující levý rozklad.

2. **Porovnání:** $(ax, a\alpha, \pi) \Rightarrow (x, \alpha, \pi)$, pokud $a \in \Sigma$. Totožné symboly na vrcholu zásobníku a ve vstupním řetězci se smažou resp. přečtou ze vstupu.
3. **Přijetí:** konfigurace (e, e, π) znamená, že řetězec je rozpoznán, analýza končí a π obsahuje posloupnost pravidel reprezentující levou derivaci řetězce podle G.
4. **Chyba:** ve všech ostatních případech analýza končí s chybovou signalizací.

Řešený příklad 17:

Mějme gramatiku:

$G = (\{S, A\}, \{a,b\}, S, P)$
$P: S \rightarrow_1 e,\ S \rightarrow_2 abA,$
$A \rightarrow_3 Saa,\ A \rightarrow_4 b$

Ověřme nejprve, zda neplatí podmínky pro LL(1) gramatiku.

Musíme tedy určit FOLLOW(S)={a, e}. Ale zároveň platí, že FIRST(abA)={a} – pravidla obsahují FIRST-FOLLOW kolizi. Není to tedy určitě silná LL(1) gramatika. Zkusme tedy nyní ověřit zda je silná LL(2).

Určíme $FOLLOW_2(S)$.

$N_e = \{S\}$
$F=\{S \rightarrow S._1\}$
$F=\{S \rightarrow S._1,\ A \rightarrow S._1aa\}$
$F=\{S \rightarrow S._1,\ A \rightarrow S._1aa,\ A \rightarrow S._1a._2a\ \}$
$FOLLOW_2(S)=\{aa, e\}$

Dále potřebujeme určit $FOLLOW_2(A)$.

$N_e = \{S\}$
$F=\{A \rightarrow A._1\}$
$F=\{A \rightarrow A._1,\ S \rightarrow abA._1\}$
$F=\{A \rightarrow A._1,\ S \rightarrow abA._1, A \rightarrow S._1aa\ \}$
$F=\{A \rightarrow A._1,\ S \rightarrow abA._1, A \rightarrow S._1aa,\ A \rightarrow S._1a._2a\ \}$

$FOLLOW_2(A)=\{aa, e\}$

Nyní prověříme podmínku pro první dvě pravidla:

$FIRST_2(eFOLLOW_2(S)) \cap FIRST_2(abAFOLLOW_2(S)) =$
$FOLLOW_2(S) \cap FIRST_2(ab) = \{aa, e\} \cap \{ab\} = \varnothing.$

$FIRST_2(SaaFOLLOW_2(A)) \cap FIRST_2(bFOLLOW_2(A)) =$
$FIRST_2(Saa) \cap FIRST_2(bFOLLOW_2(A)) = \{ab, aa\} \cap \{b, ba\} = \varnothing$.

Je to tedy silná LL(2) gramatika.

Můžeme sestrojit rozkladovou tabulku.

M	aa	ab	a	ba	bb	b	e
S	e, 1	abA, 2					e, 1
A	Saa, 3	Saa, 3		b, 4		b, 4	

Nyní zanalyzujme slovo ababbaa s vyznačením řetězce u tučným písmem (k nebo méně symbolů z dosud nepřečteného slova pro LL(2)).

(**ab**abbaa, S, e) ⇒ (**ab**abbaa, abA, 2) ⇒ (**ba**bbaa, bA, 2) ⇒ (**ab**baa, A, 2)
⇒ (**ab**baa, Saa, 23) ⇒ (**ab**baa, abAaa, 232) ⇒ (**bb**aa, bAaa, 232)
⇒ (**ba**a, Aaa, 232) ⇒ (**ba**a, baa, 2324) ⇒ (**aa**, aa, 2324) ⇒ (**a**, a, 2324)
⇒ (e, e, 2324)

Slovo bylo rozpoznáno a levá derivace je reprezentována řetězcem 2324.

6.3. Slabé LL(k) gramatiky

Kromě toho, že existují poměrně jednoduše analyzovatelné silné LL(k) gramatiky, jsou zde také **gramatiky slabé LL(k).** Jejich syntaktická analýza už není možná pouze s informací o k následujících symbolech v řetězci, ale vyžadují také informaci o dosavadním průběhu analýzy. To celou SA velmi komplikuje a vyžaduje to nejen existenci rozkladové tabulky, ale také informace a průběhu SA. Tu reprezentuje takzvaný položkový automat.

Uvažujme jednoduchý příklad, který osvětlí problém SA slabých LL(k) gramatik.

Řešený příklad 18:

Mějme gramatiku:

G = ({S, A},{a,b}, S, P)
P: $S \rightarrow_1$ aAaa, $S \rightarrow_2$ bAba,
$A \rightarrow_3$ b, $A \rightarrow_4$ e

Pokud máme v této gramatice provést expanzi symbolu A a řetězec, který následuje je ba, neumíme se rozhodnout, zda použít pravidlo 3 nebo 4, protože výsledek je na k-symbolů stejný. Jediná možnost, jak toho

rozhodnutí provést je vědět, zda jsme v předchozí analýze provedli vygenerování a podle pravidla 1 nebo b podle pravidla 2. To však už vyžaduje „pamatovat si“, co se stalo v předchozím průběhu analýzy. Pokud byl přečetný symbol a, tak se použije pravidlo 3 a pokud b, tak se použije pravidlo 4.
Tato gramatika tedy nemůže být silná LL(2), protože platí:
$FIRST_2(bFOLLOW_2(A)) \cap FIRST_2(FOLLOW_2(A)) = \{ba\}$.

Přesto jde o takzvanou slabou LL(2) gramatiku podle následující definice.

(Slabá) LL(k) gramatika

Definice 13: BKG G=(Π,Σ,S,P) se nazývá (slabá) LL(k) gramatika, pro k>0, jestliže platí pro dvě levé derivace:

$$S \Rightarrow^* wA\alpha \Rightarrow^* w\beta\alpha \Rightarrow^* wx$$
$$S \Rightarrow^* wA\alpha \Rightarrow^* w\gamma\alpha \Rightarrow^* wy$$

takové, že $FIRST_k(x) = FIRST_k(y)$, platí $\beta = \gamma$.

Jinak řečeno, gramatika G je obecná LL(k), pokud pro vygenerování řetězce začínajícího stejnými k symboly, můžeme v gramatice použít pouze jedno pravidlo (tedy podmínka jednoznačnosti expanze). Ovšem tato podmínka je **slabší** než podmínka pro silnou LL(k), protože nemusí jednoznačnost splňovat samotné pravidlo, ale může to záviset na celém odvození slova. Historie syntaktické analýzy je nejjednodušeji reprezentovaná obsahem zásobníku – tento řetězec se nazývá **perspektivní přípona**.

Definice 14: Mějme levou derivaci $S \Rightarrow^* wA\alpha \Rightarrow^* w\beta\alpha$ v G=(Π,Σ,S,P). řetězec γ nazýváme perspektivní příponou v G, pokud γ=S nebo γ je příponou řetězce $\beta\alpha$.

Úplná perspektivní přípona je taková, která začíná neterminálem. V okamžiku, kdy je v zásobníku úplná přípona, provede SA expanzi na základě této přípony.

Řešený příklad 19:

Mějme gramatiku z předchozího příkladu. Pak provedeme porovnání v SA řetězce bba a abaa, jejichž průběh se liší právě v perspektivní příponě na zásobníku.

Vstupní řetězec	Obsah zásobníku	Provedená operace
bba	S	Expanze na bAba
bba	bAba	Porovnání b
ba	Aba	Expanze na e
ba	ba	Porovnání b
a	a	Porovnání a
e	e	Přijetí

Vstupní řetězec	Obsah zásobníku	Provedená operace
abaa	S	Expanze na aAaa
abaa	aAaa	Porovnání a
baa	Aaa	Expanze na b
baa	baa	Porovnání b
aa	aa	Porovnání a
a	a	Porovnání a
e	e	Přijetí

Z uvedených tabulek můžeme vyčíst, že expanze na třetím řádku se provede podle toho, zda na zásobníku je řetězec Aba nebo Aaa. V prvním případě se expanduje na e, protože chceme vygenerovat ba a v druhém na b, protože generujeme baa. Tabulka i algoritmus SA by se tedy museli rozhodovat na základě bohatší informace než je jen neterminál a řetězec následujících symbolů. V řádcích by byly nikoliv pouze neterminály, ale celé perspektivní přípony.

M	aa	ab	ba	bb
S	aAaa, 1	aAaa, 1		bAba, 2
Aaa	e, 4		b, 3	
Aba			e, 4	b, 3

Tento postup pro náš daný jednoduchý případ lze aplikovat. Problém je, že není obecný, neboť perspektivních přípon může být nekonečně mnoho. Nekonečná tabulka samozřejmě nebude umožňovat efektivní syntaktickou analýzu. Druhým problémem je, jak zjistit pro perspektivní přípony jim příslušné pravidla.

V tomto textu se již dále zabývat SA pro slabé LL(k) gramatiky nebudeme. Z praktického hlediska to není příliš zajímavé (aplikační problémové úlohy jako jsou programovací jazyky lze řešit zápisem pomocí silných LL(k), resp. LL(1) gramatik s využitím zástupných jednosymbolových zápisů (nebo dobrou lexikální analýzou při předzpracování). Navíc existuje algoritmus, který libovolnou slabou LL(k) gramatiku převede na silnou LL(k). Jelikož **jazyk perspektivních přípon tvoří regulární jazyk**, vede SA pro slabé LL(k) gramatiky na vytvoření tzv. **charakteristického konečného automatu**, který se pak používá při SA. Vytvoření automatu zase vyžaduje aplikaci algoritmu na vytvoření **souboru tzv. LL(k) položek** a teprve s těmito informacemi můžeme provádět deterministickou SA. Celý postup je oproti předchozím algoritmům poměrně složitý a zdlouhavý. Čtenáři s hlubším zájmem o tato spíše teoreticky zajímavá témata lze doporučit literaturu [Ch84], [Ce92].

Nejdůležitější probrané pojmy:

- silné LL(k) gramatiky a (slabé) LL(k) gramatiky
- Funkce $FIRST_k$ a $FOLLOW_k$
- SA pro silné LL(k) gramatiky

Úkol k textu:

Sestrojte BKG generující nekonečný jazyk, která je silná LL(3) a zároveň není silná LL(2) gramatika. Dokažte splnění těchto podmínek a následně vytvořte rozkladovou tabulku a analyzujte libovolné slovo délky 4.

7. LL gramatiky, jazyky a jejich transformace

Cíl:

Po prostudování této kapitoly pochopíte:
- vlastnosti LL gramatik a jazyků
- zobecnění principů LL jazyků

Naučíte se:
- transformovat gramatiky z předcházejících kapitol
- využít tyto transformace na problémových aplikačních úlohách pro snadnější manipulaci s gramatikami a jazyky

I když jsme v předcházejících kapitolách kladli důraz především na algoritmy (postupy), jak provádět deterministickou – tedy implementovatelnou – SA, zavedli jsme rovněž mnoho nových tříd jazyků. Jednalo se postupně o třídu jazyků generovaných SLL(1) gramatikami, q-gramatikami, LL(1) gramatikami, silnými a slabými LL(k) gramatikami. Už v minulém díle opory jste poznali, že vlastnosti tříd jazyků jsou důležité, neboť vám dávají možnost poznat a uvědomit si principy a smysl jejich použití. Příkladem mohou být uzávěrové vlastnosti regulárních a bezkontextových jazyků nebo Chomského hierarchie jazyků. Podobné vlastnosti nyní budeme stručně zkoumat (pouze se slovní formulací myšlenky důkazu) u LL jazyků. Věnujte, prosím, této kapitole rovněž vlekou pozornost. I když se vám může zdát na první pohled méně důležitá, naopak její význam je pro pochopení a přehled o celém učivu klíčový.

7.1. Vlastnosti LL jazyků

Nejprve musíme přesně definovat, co to vlastně je LL jazyk, i když o tom asi již máte jistou představu.

Definice 15: Bezkontextový jazyk L se nazývá LL(k) jazyk, pokud existuje LL(k) gramatika G, taková že L = L(G). Bezkontextový jazyk L se nazývá LL jazyk, pokud existuje LL(k) gramatika G pro nějaké k ≥ 0 taková, že L = L(G).

LL jazyky

To jestli je nějaký jazyk LL resp. LL(k) jazyk tedy závisí na existenci příslušného typu gramatiky generující daný jazyk.

První vlastností, která je důležitá pro SA je jednoznačnost ve smyslu definice z prvního dílu opory.

Věta 3: Každá LL(k) gramatika je jednoznačná.

Jednoznačnost gramatiky je dána existencí pouze jedné levé derivace pro každé slovo jazyka. Jelikož pro každou LL(k) gramatiku platí podmínka Definice 13: musí každé odvození být jednoznačně určeno předponou řetězce na k-symbolů.

Při konstrukci LL(k) gramatiky je třeba dodržet základní podmínku a tou je vlastnost, že nemůže být zleva rekurzivní. Zleva rekurzivní je taková gramatika, která umožňuje z neterminálu A generovat řetězec, který začíná jím samým tedy opět A.

Věta 4: Žádná LL(k) gramatika není zleva rekurzivní.

Pokud by gramatika byla zleva rekurzivní, pak umožňuje generovat sekvence $A \Rightarrow^* A\alpha$. α se může přepsat buď na prázdné slovo a v tom případě tedy můžeme A odvodit různými derivacemi nebo na terminální slovo v a A na terminální slovo u, pak můžeme v různých odvozeních odvození generovat stále znovu slovo začínající na uv^{k+i}. To bylo ve sporu s jednoznačností gramatiky, protože každá LL(k) gramatika je jednoznačná. Je-li tedy gramatika zleva rekurzivní (což je z pravidel ihned zjistitelné) zjevně nemůže být LL(k).

Další vlastnost souvisí s pojmy algoritmické rozhodnutelnosti. Jde o vlastnosti, které blíže zkoumá teorie vyčíslitelnosti – čtenář se může seznámit s oporou [Pa02]. Pro naše účely tento pojem zjednodušíme do programátorské roviny (není problém si udělat paralelu mezi konkrétním programem v třeba v Pascalu a algoritmem zapsaným jiným způsobem). Představte si, že máte napsat program (algoritmus), který pro nějaké objekty řekne zda platí jistá vlastnost nebo ne. Musí to tedy fungovat pro jakýkoliv objekt dostanete na vstup a vždy musíte dostat na výstupu jasnou odpověď ano nebo ne. Pokud takový algoritmus existuje, říkáme že problém je rozhodnutelný. V opačném případě se jedná o problém nerozhodnutelný. Příkladem rozhodnutelného problému může být existence reálných kořenů pro kvadratickou rovnici. Jistě byste dokázali napsat velmi jednoduchý program, který by pro dané koeficienty kvadratické rovnice a,b,c (objekt) dokázal obecně spočítat determinant a pokud by byl nezáporný, vrátili byste odpověď ANO a v opačné případě NE.

Věta 5: Pro danou BKG a dané pevné $k \geq 0$ je rozhodnutelné, zda gramatika je LL(k) nebo ne.

Je zřejmé, že je jednoduché ověřit zda daná gramatika je silná LL(k). Stačí si spočítat příslušné množiny $FIRST_k$ a $FOLLOW_k$ a pak ověřit podmínky. Složitější je situace u slabých LL(k), ale i zde je možné vytvořit soubor LL(k) položek, vytvořit rozkladovou tabulku a pokud tato tabulka nikde neobsahuje dvě expanze pro jednu buňku, pak je LL(k).

Věta 6: Pro danou BKG je nerozhodnutelné, zda je LL(k) pro nějaké k≥0.

Postup řešení tohoto problému, který nás asi napadne jako první, je zkoušet zda gramatika je LL(1) a pokud ne, zkusit zda je LL(2) a tak dále… Pokud skutečně gramatika pro nějaké k je LL(k), pak to zjistíme (viz předchozí věta). Problém této myšlenkové konstrukce je, že nebude fungovat pokud gramatika není LL(k) pro žádné k. Pak se vlastně postup zacyklí do nekonečné smyčky a stále bude zvyšovat k do nekonečna. Postup už nám tedy nedá spolehlivě odpověď. Uvědomme si, že tato vlastnost je zásadní pro práci s LL tvary gramatik. Pokud dostaneme gramatiku, nejsme schopni jednoznačně pro každý případ ověřit, zda gramatika je vůbec LL(k) gramatika. Ještě horší je však vlastnost následující.

Věta 7: Pro danou BKG G, která není LL(k) pro dané pevné k, je nerozhodnutelné, zda k ní existuje ekvivalentní gramatika, která je LL(k).

Nemáme tedy jistotu, že obecnou BKG můžeme převést vždy na LL(k) gramatiku. Samozřejmě tento pesimistický teoretický výsledek nás ještě nemusí odradit od úsilí, převést obecnou BKG na LL(k) nebo dokonce LL(1) gramatiku. Už v předchozím textu jste viděli, že pro stejný nebo podobný problém lze někdy sestrojit různé typy gramatik. Pro převody na LL(k) gramatiky existuje několik technik, které jsme při konstrukci gramatik už v některých případech používali. Není samozřejmě s ohledem na předchozí věty zaručeno, že budou fungovat vždy, ale pro většinu praktických aplikačních úloh vedou k cíli.

Další zajímavou vlastností je, že u LL(1) gramatik je jedno zda aplikujeme podmínku silné nebo slabé LL(k) gramatiky – jde o ekvivalentní podmínky.

Věta 8: BKG je silnou LL(1) gramatikou právě tehdy, když je (slabou) LL(1) gramatikou.

Dále je jasné, že pokud gramatika splňuje podmínky pro LL(k) gramatiku, splňuje zároveň i podmínky pro LL(k+1) gramatiku. To znamená, že když je gramatika jednoznačná na k symbolů už je jednoznačná na libovolně mnoho symbolů, kterých je více než k. Zároveň platí, že každá silná LL(k) gramatika je i slabá LL(k), ale naopak ne (viz příklad slabé a silné LL(2) gramatiky) z předchozí kapitoly.

Věta 9: Pokud je BKG silná LL(k) gramatika, pak je i slabá LL(k).

Věta 10: Pokud je BKG LL(k) gramatika, pak je i LL(k+1).

Tato vlastnost vytváří hierarchii LL(k) jazyků, kde jsou do sebe postupně vnořeny třídy LL(1), LL(2), … LL(k), …jazyků navíc ještě každá třída se skládá z vnořené třídy silných LL(k) jazyků.

Poměrně elegantně lze zapsat schéma pro gramatiku, která je LL(k) a není LL(k-1).

Řešený příklad 20:

$G = (\{S, A\},\{a,b\}, S, P)$
$P: S \rightarrow aT, A \rightarrow c, A \rightarrow bB, T \rightarrow SA, T \rightarrow A, B \rightarrow b^{k-1}d, B \rightarrow e$

Tato gramatika je LL(k) lze rozhodnout na základě k následujících symbolů ve slově, zda použít pravidlo $B \rightarrow b^{k-1}d$, kde je na k-tém místě d nebo zda se použije pravidlo s epsilon a tím se dá možnost vygenrovat sekvenci symbolů b pomocí $A \rightarrow bB$. Ale na k-1 symbolů už nejsme schopni toto rozhodnutí vždy provést.

Velmi jednoduše lze také podat příklad jazyka, který není LL(k) pro žádné k. Jde o jazyk $L = \{a^n b^n | \ n \geq 0 \} \cup \{a^n c^n | \ n \geq 0 \}$.

U tohoto jazyka nelze sestrojit gramatiku, která by obecně pro počáteční symboly a dokázala jednoznačně derivovat buď na ukončovací b nebo c. Symbolů a může být na počátku potenciálně libovolný počet, což neumožňuje sestrojit obecně gramatiku pro všechna slova tohoto jazyka.

7.2. Transformace na LL gramatiky

Přestože jsme v minulé podkapitole konstatovali, že existují jazyky, ke kterým nelze sestrojit LL(k) gramatiku pro žádné k, existují rovněž jednoduché transformační techniky, které umožňují v některých případech převést gramatiku na LL(k), resp. LL(1) gramatiku. Dalším převodem, který lze dokonce provést univerzálně, je transformace LL(1) gramatiky na q-gramatiku. Všechny uvedené převodní techniky formulujeme jak teoreticky, tak vyzkoušíme na praktických příkladech.

Při převodu LL(1) gramatiky na q-gramatiku využíváme tzv. techniku rohové substituce. U LL(1) gramatiky je kolidujícím faktorem ve vztahu ke q-gramatice možnost existence neterminálu na počátku pravidla. Tohoto faktoru se lze zbavit, pokud postupně dosadíme (substituujeme) řetězce za tyto neterminály, tak že začneme od řetězců, které již začínají terminálem. Nejprve si popišme vlastní rohovou substituci.

Rohová substituce

Věta 11: Mějme BKG $G=(\Pi,\Sigma,S,P)$ a pravidlo $A \rightarrow B\alpha$ v P, kde $B \in N$ a $B \rightarrow \beta_1 | \ldots | \beta_n$ jsou všechna pravidla pro B. Vytvoříme gramatiku $G_1=(\Pi,\Sigma,S,P_1)$ vyloučením pravidla $A \rightarrow B\alpha$ a přidáním

pravidel $A \rightarrow \beta_1\alpha|\ldots|\beta_n\alpha$. Pak platí $L(G) = L(G_1)$ a pokud G je LL(k) gramatika, pak i G_1 je LL(k) gramatika. Tato transformace se nazývá rohová substituce.

Postupnou aplikací rohové substituce na uspořádané neteriminály v pořadí, kde žádný neterminál s vyšším indexem nepřipisuje na řetězec začínající neterminálem s nižším indexem, dojdeme až ke q-gramatice, pokud původní gramatika byla LL(1).

Věta 12: Pokud BKG G=(Π,Σ,S,P) je LL(1) gramatika, pak existuje q-gramatika G'=(Π,Σ,S,P') a platí, že L(G) = L(G').

Algoritmus 9: Převod LL(1) gramatiky na q-gramatiku.

Převod na q-gramatiku

Vstup: LL(1) gramatika G=(Π,Σ,S,P).
Výstup: q-gramatika G'=(Π,Σ,S,P'), kde L(G) = L(G').

Metoda:

1. Zavedeme uspořádání neterminálů, aby platilo podmínka: $A_i \rightarrow A_j\alpha$, pak $i < j$. A tedy $N=\{A_1, \ldots, A_n\}$.(Toto uspořádání lze provést pro každou LL(1) gramatiku.)
2. Položíme $i = n - 1$ a P' = P.
3. Pokud i = 0, pak máme gramatiku G', jinak pokračujeme bodem 4.
4. Každé pravidlo $A_i \rightarrow A_j\alpha$ v P', kde $j>i$, nahradíme pravidly $A_i \rightarrow \beta_1\alpha|\ldots|\beta_n\alpha$, kde $A_j \rightarrow \beta_1|\ldots|\beta_n$ (rohová substituce).
5. Pokud všechna přidaná pravidla začínají terminálem, pokračujeme krokem 6., jinak se vrátíme na krok 4.
6. Nechť i = i –1 a pokračuje krokem 3.

Aplikujme nyní tento algoritmus na gramatiku z předchozího textu.

Řešený příklad 21:

G = ({S, A, B},{x, +, (,)}, S, P)
P: $S \rightarrow_1 BA$, $A \rightarrow_2 + BA$, $A \rightarrow_3 e$, $B \rightarrow_4 x$, $B \rightarrow_5 (S)$

Krok 1: Neterminály uspořádáme například takto
$A_1 = S$, $A_2 = A$, $A_3 = B$.
Krok 2: i = 2, P = P'.
Krok 3: pokračujeme 4.
Krok 4: neexistuje pro A žádné pravidlo vyhovující dané podmínce.
Krok 5: viz 4., pokračujeme 6.
Krok 6: i = 1, pokračujeme 3.
Krok 3: pokračujeme 4.
Krok 4: provedeme rohovou substituci $S \rightarrow_1 BA$ nahradíme v P' $S \rightarrow_{1a} xA$ a $S \rightarrow_{1b} (S)A$.
Krok 5: vše začíná terminálem a pokračujeme 6.

Krok 6: i = 0, pokračujeme 3.
Krok 3: i = 0 a tedy máme q-gramatiku s pravidly:

P': $S \rightarrow_{1a} xA$ a $S \rightarrow_{1b} (\ S\ )A$, $A \rightarrow_2 + BA$, $A \rightarrow_3 e$, $B \rightarrow_4 x$, $B \rightarrow_5 (\ S\ )$

Tvar q-gramatiky může být v některých případech výhodnější pro SA než tvar LL(1) gramatiky a někdy tomu může opačně. Nevýhodou LL(1) gramatiky je delší odvození, protože dochází k postupným přepisům přes neterminály (viz příklad). Naopak výhodou je větší přehlednost gramatiky díky nižšímu počtu pravidel. V duálním pohledu se stejně můžeme dívat na q-gramatiku, která navíc může zredukovat některé neterminály, ovšem za cenu možnosti vzniku velkého množství pravidel a tím nepřehlednosti gramatiky. U praktických aplikací musíte tedy sami zvážit, co je pro vás prioritou v konkrétním případě.

Převody na LL(1) gramatiky z obecných bezkontextových gramatik nelze samozřejmě realizovat vždy (jak už vyplynulo z dřívějšího textu). Existují ale techniky, které v některých případech toto umožňují. Samozřejmě, že jich existuje více než zde uvedeme - jde o následující vybrané metody:

1. Odstranění levé rekurze.
2. Levá faktorizace.
3. Rohová substituce.
4. Pohlcení terminálního symbolu.
5. Pohlcení řetězce.

Ad 1.

Odstranění levé rekurze je operace při které se snažíme zabránit situaci, která se v žádném případě nemůže u LL(1) gramatiky vyskytnout a to je, že neterminál přepisuje na řetězec začínající jím samým. Pomocí následující věty lze tuto situaci odstranit zavedením nového neterminálu.

Odstranění levé rekurze

Věta 13: Nechť $G=(\Pi,\Sigma,S,P)$ je bezkontextová gramatika. Nechť $\{X\rightarrow X\alpha_1, X\rightarrow X\alpha_2, \ldots X\rightarrow X\alpha_r\}$ ($\alpha_i \in (\Pi\cup\Sigma)^*$, $1 \le i \le r$) je množina pravidel s levou rekurzí, ve kterých se na pravé straně úplně vlevo nachází neterminál X. Nechť $\{X\rightarrow \beta_1, X\rightarrow \beta_2, \ldots X\rightarrow \beta_s\}$ ($\beta_i \in (\Pi\cup\Sigma)^*$ jsou zbývající pravidla pro X mající na levé straně neterminál X. Nechť $G_1=(\Pi\cup\{Z\},\Sigma,S,P_1)$ je bezkontextová gramatika, která vznikla přidáním neterminálu Z k Π a dál nahrazením všech pravidel s levou rekurzí pravidly:
$X\rightarrow\beta_i$, pro $1 \le i \le s$
$X\rightarrow\beta_i Z$, pro $1 \le i \le s$
$Z\rightarrow\alpha_i$, pro $1 \le i \le r$
$Z\rightarrow\alpha_i Z$, pro $1 \le i \le r$
Pak $L(G_1)=L(G)$. Tuto operaci nazveme odstranění levé rekurze.

Poznámka: Uvědomme si, že všechna pravidla s levou rekurzí gramatiky G generují regulární množinu
$\{\beta_1,\beta_2,...\beta_s\}\{\alpha_1,\alpha_2,...\alpha_r\}^*$
což je právě množina generovaná v G_1 pravidly, která mají na levé straně neterminál X nebo Z. Vlastně nahradíme pomocí nového neterminálu Z levou rekurzi, která umožňuje generovat iteraci řetězců α_i převedením na rekurzi, která ovšem již není levou rekurzí, protože Z není prvním neterminálem v pravidlech se Z na levé straně.

Ad 2.

Levá faktorizace je operace, při která se snažíme zabránit situaci, kdy nám dvě nebo více pravidel začínají stejným řetězcem. To samozřejmě znamená, že FIRST těchto pravidel není disjunktní množina a tedy, že to nemůže být LL(k) gramatika. Odstranění je poměrně jednoduché a spočívá v jakémsi „vytknutí“ tohoto řetězce a zavedení nového neterminálu, který bude přepisovat na zbytky řetězců. To samozřejmě může buď vyřešit situaci anebo přinést s sebou novou kolizi (FIRST-FIRST nebo FIRST-FOLLOW).

Věta 14: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika.
Nechť $\{X\rightarrow \alpha\alpha_1, X\rightarrow \alpha\alpha_2, ... X\rightarrow \alpha\alpha_r\}$ ($\alpha_i \in (\Pi\cup\Sigma)^*$, $1 \le i \le r$) je množina pravidel s stejným řetězcem na začátku. Nechť $G_1=(\Pi\cup\{Z\},\Sigma,S,P_1)$ je bezkontextová gramatika, která vznikla přidáním neterminálu Z k Π a dál nahrazením všech pravidel se stejným řetězcem na začátku pravidla:
$X\rightarrow\alpha Z$,
$Z\rightarrow\alpha_i$, pro $1 \le i \le r$
Pak $L(G_1)=L(G)$. Tato operace se nazývá levá faktorizace.

Levá faktorizace

Ad 3.

Rohovou substituci jsme již probrali v rámci převodu na q-gramatiku.

Ad 4.

Pohlcení terminálního symbolu nám pomáhá vyřešit situaci, kdy za neterminálem následuje řetězec, kterým může některé pravidlo od tohoto neterminálu začínat. Řeší tedy FIRST-FOLLOW kolizi a to tak, že kolidující terminál spojí s tímto neterminálem, čímž vznikne nový neterminál reprezentující toto spojení.

Pohlcení terminálu

Věta 15: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika.
Nechť $X\rightarrow \alpha Ba\beta$ je pravidlo, kde za neterminálem B následuje terminál a a pro B platí, že $B\rightarrow \alpha_1, X\rightarrow \alpha_2, ... X\rightarrow \alpha_r$. Nechť $G_1=(\Pi\cup\{Z\},\Sigma,S,P_1)$ je bezkontextová gramatika, která vznikla

přidáním neterminálu [Ba] k Π a dál nahrazením pravidla $X \rightarrow \alpha Ba\beta$ souborem pravidel:
$X \rightarrow \alpha[Ba]\beta$,
$[Ba] \rightarrow \alpha_i a$, pro $1 \leq i \leq r$
Pak $L(G_1)=L(G)$. Tato operace se nazývá pohlcení terminálního symbolu.

Ad 5.

Podobnou operací jako je pohlcení terminálu je pohlcení celého řetězce. Tuto operaci vzhledem k podobnosti s 4. nebudeme formalizovat.

Nyní se pokusme vybrané operace demonstrovat na příkladech.

Řešený příklad 22:

Mějme gramatiku pro generování seznamu abstraktních příkazů a bloků ve složených závorkách oddělených středníkem.

G = ({S, A},{p, ; , {, } }, S, P)
P: $S \rightarrow_1 A$, $S \rightarrow_2 S;A$,
$A \rightarrow_3 p$, $A \rightarrow_4 \{S\}$

Tato gramatika nemůže být LL(1), neboť obsahuje levou rekurzi. Provedeme její převod pomocí odstranění levé rekurze. Zavedeme nový neterminál Z a zrušíme pravidlo s levou rekurzí.

G = ({S, A, Z},{p, ; , {, } }, S, P)

$S \rightarrow_1 A$, $S \rightarrow_2 AZ$,
$A \rightarrow_3 p$, $A \rightarrow_4 \{S\}$,
$Z \rightarrow_5 ;AZ$, $Z \rightarrow_6 ;A$.

Tato gramatika již neobsahuje levou rekurzi, ovšem obsahuje dvě FIRST-FIRST kolize u neterminálu S a Z. Pokusíme se je tedy odstranit levou faktorizací. Zavedeme nové neterminály X a Y, které vzniknou její aplikací.

G = ({S, A, Z, X, Y},{p, ; , {, } }, S, P)

$S \rightarrow_1 AX$,
$A \rightarrow_2 p$, $A \rightarrow_3 \{S\}$,
$Z \rightarrow_4 ;AY$,
$X \rightarrow_5 e$, $X \rightarrow_6 Z$,
$Y \rightarrow_7 Z$, $Z \rightarrow_8 e$.

U pravidel 1. – 4. není již žádná kolize. Avšak u neterminálu X a Y může být kolize FIRST-FOLLOW. Nejprve spočítáme FOLLOW(X) = { } , e} a FOLLOW(Y) = { }, e }.

Platí, že FOLLOW(X) ∩ FIRST(Z) = { }, e } ∩ { ; } = ∅ a rovněž FOLLOW(Y) ∩ FIRST(Z) = { }, e } ∩ { ; } = ∅.

Gramatika tedy je LL(1) gramatika.

Existují ale samozřejmě i případy, kdy nám levá faktorizace nezaručí, že dostaneme LL(1) gramatiku.

Řešený příklad 23:

G = ({S, A, B},{a, x, y, z }, S, P)
P: S $\rightarrow_1$ aAxx, S $\rightarrow_2$ aByy, S $\rightarrow_3$ zy, S $\rightarrow_4$ zx,
A $\rightarrow_5$ aAx, A $\rightarrow_6$ z, B $\rightarrow_7$ aBy, B $\rightarrow_8$ z

Po levé faktorizaci dostaneme gramatiku:

G = ({S, A, B, X, Y},{a, x, y, z }, S, P)
P: S $\rightarrow_1$ aX, S $\rightarrow_2$ zY,
X $\rightarrow_3$ Axx, X $\rightarrow_4$ Byy,
Y $\rightarrow_5$ x, Y $\rightarrow_6$ y,
A $\rightarrow_7$ aAx, A $\rightarrow_8$ z, B $\rightarrow_9$ aBy, B $\rightarrow_{10}$ z

Tato gramatika není LL(1) gramatika, protože
FIRST(Axx)∩FIRST(Byy) = {a, z}.

V určitých případech je potřeba před vlastní faktorizací provést ještě rohovou substituci.

Řešený příklad 24:

G = ({S, A, B},{a, b, c, d}, S, P)
P: S → aA, S → BA,
A → cA, A → d, B → aB, B → bA

Tato gramatika není LL(1), protože FIRST(aA)∩FIRST(BA) = {a} a taktéž nemůžeme provést faktorizaci. Abychom faktorizaci symbolu a mohli provést, je potřeba nejprve provést rohovou substituci B.

S → aA, S → aBA, S → bAA,
A → cA, A → d, B → aB, B → bA

Nyní se již může faktorizovat zavedením nového neterminálu X.

G = ({S, A, B, X},{a, b, c, d}, S, P)

S → aX, S → bAA,
X → A, X → BA,
A → cA, A → d, B → aB, B → bA

Tato gramatika už je LL(1).

Nyní se zaměříme na odstraňování kolize FIRST-FOLLOW pomocí pohlcování symbolů.

Řešený příklad 25:

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

S → AaB, A → e, A → aaB,
B → c, B → bB

FIRST(aaB)∩FOLLOW(A) = {a} a tedy nejde o LL(1) gramatiku.

Tuto kolizi můžeme odstranit pomocí pohlcení terminálu a, který kolizi způsobuje jeho pohlcením neterminálem A.

G = ({S, A, B, [Aa]},{a, b, c}, S, P)

S → [Aa]B, A → e, A → aaB,
B → c, B → bB, [Aa] → a, [Aa] → aaBa

Tím ovšem vznikla FIRST-FIRST kolize posledních dvou pravidel, kterou musíme dále řešit pomocí faktorizace zavedením X. Také lze zredukovat neterminál A, protože se již v gramatice nedá od S přepsat.

G = ({S, B, [Aa], X},{a, b, c}, S, P)

S → [Aa]B, B → c, B → bB, [Aa] → aX, X → e, X → aBa

V této gramatice FOLLOW(X) = {b, c} a tedy není zde FIRST-FOLLOW kolize.

Vhodnou kombinací transformačních technik tedy můžete (ale nemusíte) dospět k LL(k), resp. LL(1) gramatice.

Nejdůležitější probrané pojmy:
- hierarchie LL(k) jazyků
- transformační techniky (odstranění levé rekurze, rohová substituce, levá faktorizace, pohlcení terminálu, pohlcení řetězce)

Úkol k textu:

1. Sestrojte BKG pro syntaktickou strukturu formule predikátové logiky, kde se mohou vyskytovat n-nární predikáty pojmenované symboly p, q, r a dále mohou obsahovat termy – buď vnořené funktory f, g, h s n argumenty nebo konstanty a, b, c nebo symboly pro proměnné x, y, z. Povolené logické spojky jsou konjunkce ($\wedge$), disjunkce($\vee$) a negace($\neg$) podle standardní definice predikátové logiky. Lze samozřejmě vnořovat i závorkované formule.

   Příklad: $(p(x,f(y,g(c))) \vee \neg r(b)) \wedge q(z,y)$
2. Vámi sestrojenou gramatiku převeďte na LL(1) gramatiku.
3. Sestrojte rozkladovou tabulku pro tuto LL(1) gramatiku a zanalyzujte formuli z příkladu 1.

8. Syntaktická analýza „zdola nahoru“, LR gramatiky

Cíl:

Po prostudování této kapitoly pochopíte:
- Způsob analýzy „zdola nahoru“ pomocí rozkladové tabulky
- Princip LR gramatik
- Funkce BEFORE a EFF

V minulých kapitolách a vlastně v celém textu se věnujeme především analýze „shora dolů“. Tento způsob je vhodný zejména pro vlastní přímou implementaci, protože je poměrně jednoduchý a přehledný i pro neautomatizované zpracování (což platí zejména až do třídy LL(1) gramatik). Přesto alespoň v krátkosti zmíníme o druhém způsobu deterministických analýz v lineárním čase a ten je založen na SA pomocí rozkladových tabulek způsobem „zdola nahoru“. Tento způsob je v jistém ohledu více obecný (jak udivíme u vlastností LR jazyků) a využívá se ve větší míře pro automatické generování analyzátorů pomocí počítačových programů. Platí se za to ovšem mnohem složitější konstrukcí rozkladových tabulek, což v případě automatizované tvorby není překážkou.

8.1. Model analýzy „zdola nahoru“

Analýza „zdola nahoru“

Model analýzy **„zdola nahoru“** je založen na postupných redukcích řetězců zpětně podle pravidel gramatiky a přesunech symbolů do zásobníku. V jistém smyslu jde tedy o duální postup oproti analýze „shora dolů“, kde naopak provádíme expanzi v směru pravidel. Podobně jako pro analýzu „shora dolů“ je možné uvažovat o obecném principu SA pomocí zásobníkového automatu. Takto sestrojený zásobníkový automat (potenciálně nedeterministický) by pracoval podle následujících tří typů pravidel):

1. Přesun symbolu vstupního slova na zásobník.
2. Redukce řetězce na vrcholu zásobníku zpětně podle přepisovacího pravidla.
3. Přijetí v případě, že jsme přečetli celé slovo a na zásobníku jsme vytvořili S – počáteční neterminál.

Formálně lze uvedený postup popsat jako alternativu k důkazu Věta 1:

Mějme bezkontextovou gramatiku $G=(\Pi,\Sigma,S,P)$.
Sestrojíme ZA M tak, že $L(G)=L_{PZ}(M)$.
Položíme $M = (\{p\},\Sigma,\Pi\cup\Sigma\cup\{\#\},\delta,p,\#,\varnothing)$.
Pro δ platí:
$\delta(p,a,e)=\{(p,a)\}$; $\forall a \in \Sigma$

$\delta(p,e,\alpha)=\{(p,A)|(A\rightarrow \alpha) \in P\}; \forall X \in \Pi$
$\delta(p,e, S\#)=\{(p,e)\}$

Je však nutno rozšířit definici ZA tak, aby umožňoval přepisy celých řetězců na zásobníku, nikoliv pouze symboly!

Takto sestrojený ZA má tři typy pravidel – buď přepisuje řetězec na neterminál, ukládá terminální symboly na zásobník nebo v případě, že přečte celé slovo a na zásobníku zbývá právě S a počáteční symbol #, provede přijetí slova. Obecně je automat nedeterministický – tedy musí si najít správnou cestu. Pokusme se sestrojit takový automat pro gramatiku z příkladu 1.

Řešený příklad 26:

G = ({S, A, B},{x, *, +, (,)}, S, P)
P: $S \rightarrow_1 A + S$, $S \rightarrow_2 A$
$A \rightarrow_3 B * A$, $A \rightarrow_4 B$
$B \rightarrow_5 (S)$, $B \rightarrow_6 x$

Zásobníkový automat sestrojený tímto principem bude následující:
$M = (\{p\},\Sigma,\Pi\cup\Sigma\cup\{\#\},\delta,p, \#,\varnothing)$, kde
$\Sigma =\{x, *, +, (,)\}$, $\Pi = \{S, A, B, \#\}$
δ :

1. $\delta(p,e, A + S)=(p, S)$
2. $\delta(p,e,A)=(p, S)$
3. $\delta(p,e,B * A)=(p, A)$
4. $\delta(p,e,B)=(p, A)$
5. $\delta(p,e, (S))=(p, B)$
6. $\delta(p,e,x)=(p, B)$

a dále pravidla pro přesun a ukončení
P1. $\delta(p,x,e)=(p,x)$, P2. $\delta(p,*,e)=(p,*)$, P3. $\delta(p,+,e)=(p,+)$,
P4. $\delta(p, (, e)=(p,( )$, P5. $\delta(p,), e)=(p,))$, U. $\delta(p,e,\#S)=(p,e)$

Tento zásobníkový automat můžeme dále využít pro rozpoznávání řetězce např. x * x (pozor je potřeba vzít úvahu, že řetězec čteme jako zrcadlově obrácené slovo!).

$(p, x * x, \#) \Rightarrow_{P1} (p, x * , x \#) \Rightarrow_6 (p, x * , B \#) \Rightarrow_4 (p, x * , A \#)$
$\Rightarrow_{P2} (p, x , * A \#) \Rightarrow_{P1} (p, e , x * A \#) \Rightarrow_6 (p, e , B * A \#)$
$\Rightarrow_3 (p, e , A \#) \Rightarrow_2 (p, e , S \#) \Rightarrow_U (p, e , e)$

Slovo jsme tedy úspěšně rozpoznali, ovšem analýza vykazuje opět velmi silné znaky nedeterminismu – v mnoha případech by bylo možné udělat buď přesun nebo redukci. Právě aby k tomuto nedeterminismu nedocházelo, je potřeba zavést speciální typy gramatik – LR gramatiky.

Opět budeme moci rozlišit silné a slabé LR(k) podle toho, zda k jejich SA potřebujeme znát pouze určitou část následujícího analyzovaného slova nebo i informaci o dosavadním průběhu analýzy. V tomto textu se budeme zabývat jen silnými LR(k) gramatikami – slabé LR(k) gramatiky se potýkají se stejnými problémy jako slabé LL(k) gramatiky (nutnost tvorby položek apod.)

8.2. LR(k) gramatiky a jejich syntaktická analýza

Pro definici silné LR(k) gramatiky je potřeba mít k dispozici podobné funkce jako pro LL(k) gramatiky. Z principu SA pro LR gramatik však tyto funkce fungují odlišným způsobem. Jde o funkce BEFORE (analogie s FOLLOW), která určuje které symboly předcházejí určitému neterminálu a funkci EFF „e-free FIRST“, která má význam symbolů nacházejících se na začátku řetězce.

Funkce BEFORE a EFF

Definice 16: Máme neterminál X a řetězec $\alpha \in (\Pi \cup \Sigma)^*$ v G=(Π,Σ,P,S), pak platí:

$BEFORE(X) = \{Y \mid S \Rightarrow^* \alpha Y X \beta, Y \in (\Pi \cup \Sigma)^* \} \cup \{\# \mid S \Rightarrow^* X\beta\}$

$EFF_k(\alpha) = \{w \mid w \in FIRST_k(\alpha)$ a existuje pravá derivace $\alpha \Rightarrow^* \beta \Rightarrow^* wx$ taková, že pro β neplatí $\beta = Awx\}$

Do množiny EFF tedy patří všechny řetězce z FIRST, které byly derivované derivací $\alpha \Rightarrow^* \beta \Rightarrow^* wx$ tak, že první neterminál v β nebyl nahrazený e.

Silná LR(k) gramatika

Definice 17: BKG G=(Π,Σ,S,P) se nazývá silná LR(k) gramatika, pokud pro rozšířenou gramatiku G'=(Π∪{S'},Σ,S',P∪{S'→S}) platí pro každé dvojice pravidel v P':

1.

- **a. $A \rightarrow \alpha X, B \rightarrow \beta X$,**
- **b. $A \rightarrow \alpha X, B \rightarrow e$ a $X \in BEFORE(B)$**
- **c. $A \rightarrow e, B \rightarrow e, X \in BEFORE(B)$ a $X \in BEFORE(A)$**

Pak $FOLLOW_k(A) \cap FOLLOW_k(B) = \emptyset$.

2.

- **a. $A \rightarrow \alpha X, B \rightarrow \alpha X \gamma$,**
- **b. $A \rightarrow e, B \rightarrow \alpha X \gamma$ a $X \in BEFORE(A)$**
- **c. $A \rightarrow e, B \rightarrow \gamma, X \in BEFORE(A)$ a $X \in BEFORE(B)$**

Pak $FOLLOW_k(A) \cap EFF_k(\gamma FOLLOW_k(B)) = \emptyset$.

Podmínka 1. zabezpečuje, že pro redukci je možné se rozhodnout o pravidle na základě řetězce na k symbolů (b. a c. jsou případy, kdy se jeden nebo oba řetězce vypustí a pak je nutno zkoumat jen situaci, kdy jsou předcházející řetězce totožné). Podmínka 2. určuje jednoznačnost provedení redukce nebo přesunu.

Pro tyto gramatiky lze s pomocí výše uvedených funkcí sestrojit rozkladovou tabulku, která bude mírně odlišné struktury než u LL(k) gramatik. Bude totiž obsahovat v řádcích neterminály i terminály, což vyplývá z faktu, že na zásobníku se mohou redukovat řetězce obou druhů symbolů. V jednotlivých buňkách pak budou symboly reprezentující redukce, přesuny a přijetí.

Algoritmus 10: Vytvoření rozkladové tabulky pro silnou LR(k) gramatiku.

Rozkladová tabulka pro silnou LR(k)gramatiku

Vstup: LR(k) gramatika G=(Π,Σ,S,P).
Výstup: rozkladová tabulka M pro G definovaná na $(\Pi \cup \Sigma \cup \{\#\}) \times \Sigma^{*k}$.
Postup:
Nejprve vytvoříme novou ekvivalentní gramatiku:
G'=(Π∪{S'},Σ,S',P∪{S'→S})
Rozkladovou tabulku sestrojíme podle následujících pravidel:

1. M(X, u) = redukce (i), pokud $A \rightarrow \alpha X$ je i-té pravidlo v P a $u \in FOLLOW_k(A)$.
2. M(X, u) = redukce (i), pokud $A \rightarrow e$ je i-té pravidlo v P, $X \in BEFORE(A)$ a $u \in FOLLOW_k(A)$.
3. M(S, e) = přijetí.
4. M(X, u) = přesun, pokud $B \rightarrow \beta X \gamma \in P$ a $u \in EFF_k(\gamma FOLLOW_k(B))$.
5. M(X, y) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Podmínka 1. určuje redukce pokud je X na zásobníku následuje v řetězci právě ta kombinace symbolů, která může následovat za neterminálem v daném pravidle pro redukci (A). Podmínka 2. je obdobná, ale jelikož pravidlo může vypustit A, zajímá nás, co se může vyskytnout před ním. Podmínka 3. reprezentuje situaci, že jsme celý řetězec úspěšně zredukovali až na S a 5. je situace, kdy se analýza neúspěšně zastaví. Podmínka 4. popisuje přesuny, které se mohou vykonat v případech, kdy symboly ve slově se potenciálně shodují s tím, co následuje za X (tedy takové přesuny povedou v budoucnu možná k redukci – jinak to nemá smysl).

Abychom mohli provádět syntaktickou analýzu, je potřeba ještě nadefinovat modifikovaný algoritmus SA.

Algoritmus 11: Syntaktická analýza pro silné LR(k) gramatiky

SA pro silnou LR(k)gramatiku

Vstup: rozkladová tabulka M pro silnou LR(k) gramatiku G=(Π,Σ,S,P), vstupní řetězec $w \in \Sigma^*$.

Výstup: pravý rozklad (derivace) vstupního řetězce v případě, že $w \in L(G)$, jinak chybová signalizace.

Postup:

- Algoritmus čte vstupní řetězec, používá zásobník a má k dispozici slovo u = $FIRST_k(x)$, kde x je dosud nepřečtená část řetězce, symbolem X označíme vrchol zásobníku. (vrchol zásobníku je na konci slova)
- Počáteční situace je (w, #, e).
- Vykonávají se přechody 1. a 2. dokud nenastane situace 3. nebo 4.
  1. **Redukce:** Pokud M(X, u) = redukce (i), vyloučíme ze zásobníku α, které je na pravé straně pravidla $A \rightarrow \alpha$. Pokud na zásobníku nebyl řetězec α, nastává chybová signalizace a analýza končí, jinak číslo i je připojeno k posloupnosti reprezentující pravý rozklad a do zásobníku zařadíme řetězec A.
  2. **Přesun**: Pokud M(X, u) = přesun, přečte se vstupní symbol a uloží se na vrchol zásobníku.
  3. **Přijetí:** Pokud M(X,u) = přijetí, řetězec je rozpoznán, analýza končí a π obsahuje posloupnost pravidel reprezentující pravou derivaci řetězce podle G.
  4. **Chyba:** ve všech ostatních případech analýza končí s chybovou signalizací.

Řešený příklad 27:

Mějme LR(1) gramatiku pro generování aritmetických výrazů se sčítáním, násobením, vnořenými výrazy se závorkami a operandem x:
G = ({S, A, B, C, D},{+,*,(,), x}, S, P)
P: $S \rightarrow_1 AB$, $A \rightarrow_2 S+$, $A \rightarrow_3 e$, $B \rightarrow_4 CD$, $C \rightarrow_5 B*$, $C \rightarrow_6 e$,
$D \rightarrow_7 (S)$, $D \rightarrow_8 x$

Gramatiku rozšíříme dále o pravidlo $S' \rightarrow_0 S$. Vytvoříme tabulku, kde použijeme následující zkratky: P – přesun, R(i) – redukce (i), A – přijetí.

M	x	+	*	(	)	e
S		P			P	A
A	R(6)			R(6)		
B		R(1)	P		R(1)	R(1)
C	P			P		
D		R(4)	R(4)		R(4)	R(4)
x		R(8)	R(8)		R(8)	R(8)
+	R(2)			R(2)		
*	R(5)			R(5)		
(	R(3)			R(3)		
)		R(7)	R(7)		R(7)	R(7)
#	R(3)			R(3)		

Při konstrukci této tabulky jsme použili hodnoty funkce BEFORE(A) = {#, (}, BEFORE(C) = {E‘}

Nyní zanalyzujme slovo x * x + x:

(x * x + x , # , e) ⇒ (x * x + x , #A , 3) ⇒ (x * x + x , #AC , 36) ⇒
(* x + x , #ACx , 36) ⇒ (* x + x , #ACD , 368) ⇒
(* x + x , #AB , 3684) ⇒ (x + x , #AB* , 3684) ⇒
(x + x , #AC , 36845) ⇒ (+ x , #ACx , 36845) ⇒
(+ x , #ACD , 368458) ⇒ (+ x , #AB , 3684584) ⇒
⇒ (+ x , #S , 36845841) ⇒
(x , #S+ , 36845841) ⇒ (x , #A , 368458412) ⇒
(x , #AC , 3684584126) ⇒ (e , #ACx , 3684584126) ⇒
(e , #ACD , 36845841268) ⇒ (e , #AB , 368458412684) ⇒
(e , #S , 3684584126841) ⇒ Přijetí

Slovo bylo rozpoznáno a pravá derivace je reprezentována řetězcem 3684584126841.

Kromě silných LR(k) gramatik existují ještě jejich další podtřídy a nadtřídy. Nebudeme uvádět jejich přesné definice, pouze si provedeme jejich stručný výčet.

1. **Slabé LR gramatiky** – jsou definovány analogickou podmínkou jako slabé LL gramatiky, tj. není kladeno omezení na pravidla, ale na jednoznačnost odvození. Z toho plyne mnohem obtížnější SA a konstrukce rozkladových tabulek podobně jako slabých LL(k) gramatik.
2. **LR(0) gramatiky** – využívají při SA informaci pouze o průběhu SA a nepotřebují znát žádnou část řetězce.
3. **Jednoduché LR(k) gramatiky** – tyto tzv. SLR(k) gramatiky umožňují díky omezení jednodušeji konstruovat rozkladové tabulky.

8.3. Vlastnosti LR jazyků

Podobně jako LL jazyků můžeme formulovat některé důležité vlastnosti.

LR jazyky

Definice 18: Bezkontextový jazyk L se nazývá LR(k) jazyk, pokud existuje LR(k) gramatika G, taková že L = L(G). Bezkontextový jazyk L se nazývá LR jazyk, pokud existuje LR(k) gramatika G pro nějaké k ≥ 0 taková, že L = L(G).

To jestli je nějaký jazyk LR resp. LR(k) jazyk tedy závisí na existenci příslušného typu gramatiky generující daný jazyk.

První vlastností opět zaručuje jednoznačnost.

Věta 16: **Každá LR(k) gramatika je jednoznačná.**

Věta 17: **Každá LL(k) gramatika je LR(k) gramatika.**

Zajímavou vlastností je, že každá LL(k) gramatika je LR(k) gramatika a naopak nikoliv. To znamená, že LR gramatiky jsou obecnější třídou gramatik než LL gramatiky – což jim dává větší expresivitu. Ovšem to je za cenu komplikovanější SA, resp. tvorby tabulek.

Nejdůležitější probrané pojmy:

- LR gramatiky a jazyky
- Funkce BEFORE a EFF
- Vlastnosti LR jazyků

9. Algoritmy SA a jejich implementace

Cíl:

Po prostudování této kapitoly se stručně seznámíte s metodami SA a jejich implementace (výhody a nevýhody):

- Obecné metody pro BKG – Earleyho algoritmus, CYK algoritmus
- Zásobníkový automat
- Rozkladové tabulky
- Rekurzivní sestup
- Metoda spojových seznamů

Z teoretického hlediska jsme se nyní věnovali především rozkladovým tabulkám pro LL, resp. LR gramatiky. Samozřejmě existuje mnoho způsobu pro implementaci SA a ty mají své výhody a nevýhody z hlediska **implementace na prostředcích pro automatizaci** (zejména na počítačích). Jejich důkladnější rozbor je již spíše **náplní kurzu překladače**, neboť k překladačům neoddělitelně syntaktická analýza patří jako jejich podstatná a nezbytná součást. Přesto se v následující kapitole, ale alespoň velmi stručně zmíníme o aspektech některých metod SA. Zejména půjde o možnosti **implementace datových struktur, časovou a prostorovou náročnost**.

9.1. Obecné algoritmy analýzy

Časová složitost jakékoliv metody – algoritmu – hraje v informatice klíčovou úlohu [Pa02]. Praktická realizace algoritmu musí být dostatečně „rychlá“ a nesmí spotřebovat „enormně mnoho paměti“, abychom s ní v praxi uspěli. Jistě znáte optimalizační problémy typu „Problém obchodního cestujícího“, kdy neznáme dostatečně efektivní klasický algoritmus pro jeho řešení. Samozřejmě existují různé moderní metody pro hledání optimálního řešení založené například na evolučních technikách, ale klasický deterministický algoritmus má vždy exponenciální složitost, což jej pro praxi činí nepoužitelným od určité velikosti problému (počtu měst, které má obchodní cestující navštívit).

Pro syntaktickou analýzu obecných bezkontextových gramatik existují algoritmy s mnohem lepší funkcí časové složitosti – s kubickou složitostí, resp. kvadratickou pro jednoznačné gramatiky. Prvním z nich je tzv. **Earleyho analyzátor (algoritmus).** Je založen na tečkové notaci, podobně jako algoritmy pro výpočet FIRST a FOLLOW. Jde o algoritmus z rodiny tzv. grafových algoritmů. Efektivní je zejména u gramatik s levou rekurzí. Popišme si nyní jeho způsob výpočtu.

Earleyho algoritmus

Algoritmus je založen na tečkové notaci, tedy pravidlo A $\rightarrow$ B.CD s tečkou před C reprezentuje situaci, že B již bylo analyzováno a zbývá

zanalyzovat CD. Pro každou vstupní pozici analyzovaného řetězce vytvoříme množinu stavů, které reprezentují kombinaci dvou prvků:

1. Pravidlo s tečkovou notací
2. Pozice, na které pravidlo začalo – výchozí stav.

Stav na vstupní pozici k se nazývá S(k). Počáteční stav analýzy je S(0). Analyzátor vykonává iterativně 3 typy operací:

1. Predikci: Pro každý stav v S(k) tvaru $(X \rightarrow \alpha\ .Y\ \beta, j)$, kde j je výchozí stav, přidej $(Y \rightarrow .\ \gamma, k)$ do S(k) pro každé pravidlo s Y na levé straně.
2. Čtení: Pokud máme a jako další symbol v analyzovaném řetězci, pak pro každý stav v S(k) ve tvaru $(X \rightarrow \alpha\ .a\ \beta, j)$, přidáme $(X \rightarrow \alpha\ a\ .\ \beta, j)$ do S(k+1).
3. Ukončování: Pro každý stav v S(k) tvaru $(X \rightarrow \alpha\ .,\ j)$ najdeme stavy v S(j) tvaru $(Y \rightarrow \alpha\ .\ X\ \beta, i)$ a přidáme stavy $(Y \rightarrow \alpha\ X\ .\ \beta, i)$ do S(k).

Algoritmus stále přidává nové stavy, dokud lze nové přidat. To lze implementovat například pomocí fronty ještě nevyřešených stavů. Tento postup vlastně simuluje procházení gramatiky na základě příslušných symbolů ve vstupu, podobně jako je tomu při konstrukci množiny FIRST.

Nyní se podívejme na řešený příklad, převzatý z [Wi05].

Řešený příklad 28:

Mějme gramatiku pro generování aritmetických výrazů se sčítáním a násobením a čísly jako operandy.

```
P -> S      # startovací pravidlo
S -> S + M | M
M -> M * T | T
T -> number
```

Vstupní řetězec: 2 + 3 * 4

Generované stavy:

```
== S(0): • 2 + 3 * 4 ==
(1)  P -> • S          (0)   # startovací pravidlo
(2)  S -> • S + M      (0)   # predikce z (1)
(3)  S -> • M          (0)   # predikce z (1)
(4)  M -> • M * T      (0)   # predikce z (3)
(5)  M -> • T          (0)   # predikce z (3)
(6)  T -> • number     (0)   # predikce z (5)
```

```
== S(1): 2 • + 3 * 4 ==
(1)  T -> number •    (0)   # čtení z S(0)(6)
(2)  M -> T •         (0)   # ukončování z S(0)(5)
(3)  M -> M • * T     (0)   # ukončování z S(0)(4)
(4)  S -> M •         (0)   # ukončování z S(0)(3)
(5)  S -> S • + M     (0)   # ukončování z S(0)(2)
(6)  P -> S •         (0)   # ukončování z S(0)(1)

== S(2): 2 + • 3 * 4 ==
(1)  S -> S + • M     (0)   # čtení z S(1)(5)
(2)  M -> • M * T     (2)   # predikce z (1)
(3)  M -> • T         (2)   # predikce z (1)
(4)  T -> • number    (2)   # predikce z (3)

== S(3): 2 + 3 • * 4 ==
(1)  T -> number •    (2)   # čtení z S(2)(4)
(2)  M -> T •         (2)   # ukončování z S(2)(3)
(3)  M -> M • * T     (2)   # ukončování z S(2)(2)
(4)  S -> S + M •     (0)   # ukončování z S(2)(1)
(5)  S -> S • + M     (0)   # ukončování z S(0)(2)
(6)  P -> S •         (0)   # ukončování z S(0)(1)

== S(4): 2 + 3 * • 4 ==
(1)  M -> M * • T     (2)   # čtení z S(3)(3)
(2)  T -> • number    (4)   # predikce z (1)

== S(5): 2 + 3 * 4 • ==
(1)  T -> number •    (4)   # čtení z S(4)(2)
(2)  M -> M * T •     (2)   # ukončování z S(4)(1)
(3)  M -> M • * T     (2)   # ukončování z S(2)(2)
(4)  S -> S + M •     (0)   # ukončování z S(2)(1)
(5)  S -> S • + M     (0)   # ukončování z S(0)(2)
(6)  P -> S •         (0)   # ukončování z S(0)(1)
```

Vidíte, že máme i poměrně jednoduchý algoritmus pro obecné BKG a navíc jeho časová složitost (zvláště pro jednoznačné gramatiky) je vcelku přijatelná (kvadratická složitost se považuje vzhledem k exponenciální složitosti optimalizačních problémů za zvládnutelnou). Skrytý problém této konstrukce je však v její paměťové náročnosti. Už v tomto jednoduchém příkladě je jasné, že vzniká velké množství stavů, které je potřeba ukládat v nějaké datové struktuře. Proto se zdá být rozumnější (pokud to lze) používat omezené třídy jazyků z minulých kapitol, kde máme algoritmy SA s lineární časovou složitostí a navíc bez nutnosti uchovávat potenciálně rozsáhlé datové struktury.

CYK algoritmus

Dalším z rodiny SA pro obecné BKG je známý **Cocke-Younger-Kasami (CYK) algoritmus** nebo také (CKY). Tento algoritmus ve své nemodifikované verzi dokáže rozpoznat pro každou BKG v Chomského

normální formě (každou BKG lze poměrně jednoduše převést na CHNF – viz první díl textu), zda slovo patří do jazyka generovaného touto gramatikou. Jde o implementaci metod tzv. dynamického programování (metody využívají rozklad na podproblémy a jejich optimalizaci, např. naivní výpočet Fibonnaciho čísel prostou rekurzí vs. výpočet na základě ukládání dílčích hodnot pro menší argumenty, což optimalizuje výpočet, protože není nutné znovu počítat již dříve vyčíslené hodnoty).

CKY algoritmus má rovněž v nejhorším případě kubickou časovou složitost a je důležitý zejména z teoretického hlediska, protože dává důkaz o rozhodnutelnosti problému příslušnosti slova do BKJ.

9.2. Zásobníkový automat

Zásobníkový automat je přirozeným kandidátem na roli syntaktického analyzátoru. I když dokáže rovněž analyzovat obecný BKJ, jeho nedeterministická verze vyžaduje určitou metodu simulace, abychom dostali implementovatelný deterministický postup. Právě protože je jeho myšlenka příliš jednoduchá vede obecně při deterministické simulaci na exponenciální složitost , což je pro praxi nepoužitelné. Nicméně jeho implementace je poměrně jednoduchá – jde vlastně o zásobník s pravidly, které lze zapsat do vícerozměrného pole.

9.3. Rozkladové tabulky

Rozkladové tabulky pro LL a LR gramatiky, které jsme detailně rozebírali v tomto textu mají svou výhodu v jednoduché implementaci a lineární časové složitosti analýzy, jsou-li jednou vytvořeny. Tabulku lze jednoduše implementovat jako vícerozměrné pole nebo dynamickou strukturu a pak v ní jen hledat příslušnou kombinaci symbol ve vstupní slově a symbol na vrcholu zásobníku a provést danou akci. To provádíme iterativně až do dosažení přijetí nebo chybové signalizace.
Na druhou stranu pro danou gramatiku je po sestrojení tabulky a implementaci poměrně složité provést rozšíření. Například kdybychom chtěli místo celých čísel v gramatice použít čísla reálná, znamená to přebudovat celou rozkladovou tabulku na základě nové gramatiky. Navíc je rozkladová tabulku pro člověka téměř nečitelná ve smyslu logiky daného jazyka. Kupříkladu byste asi těžko jen na základě rozkladové tabulky dokázali určit jaký jazyk generuje – u výchozí gramatiky to může být přece jen více zřejmé.
Pro ruční implementaci (myslí se tím, vlastní tvorba tabulky a její zápis a použití ve formě vlastního počítačového programu) jsou poměrně vhodné LL(1), resp. LL(k) gramatiky. Naopak tvorba LR analyzátorů je velmi pracná a nepřehledná takže pro ruční implementaci jsou nevhodné. Ale díky větší obecnosti se v praxi více používají automatické generátory (hotové profesionální aplikace) založené na LR gramatikách. Tato oblast je náplní teorie a praxe překladačů, ale pro zájemce je možné dát odkaz na

existující a dobře známé programy jako je LL-gen (pro LL gramatiky) nebo více používaný YACC či Bison.

9.4. Metoda rekurzivního sestupu

Rekurzivní sestup

Velice oblíbenou, jednoduchou a přehlednou metodou vedoucí přímo ke zdrojovému kódu je **metoda rekurzivního sestupu** (**Recursive Descent Parsing**) [Le02]. Metoda je založena na principu analýzy „shora dolů“ a je tedy blízká LL gramatikách.
Metoda rekurzivního sestupu spočívá v konstrukci procedur strukturovaného programovacího jazyka přesně dle Backusovy-Naurovy formy (BNF), kde je každému neterminálu přiřazena jedna procedura a je volána procedura GetChar (načítající vždy následující symbol slova) před každým terminálem. Výskyt neterminálu v pravidle je v proceduře nahrazen rekurzivně voláním příslušné procedury. Iterace a podmínka v Backus-Naurově formě je nahrazena jednoduše jejich programátorskými protějšky. Vyžaduje se, aby jazyk byl definován LL(k) gramatikou. Z hlediska časové složitosti jde opět o obecně neefektivní metodu s exponenciální časovou složitostí, nicméně pro jednoduché gramatiky z praxe je použitelná a zejména je její výhodnou vysoká čitelnost kódu ve vztahu k výchozí gramatice. Navíc existuje modifikace tzv. **packrat parser**, která pro omezenou třídu takzvaných **parsing expression grammars** pracuje v lineární čase. Také je tento postup flexibilní, protože umožňuje kdykoliv změnit a přidat syntaktický element bez nutnosti měnit celý kód, ale pouze dotčenou část gramatiky (například změna struktury číslo z celého čísla na reálné znamená pouze změnu procedury reprezentující tento element). Podívejme se nyní na příklad gramatiky pro generování aritmetických výrazů se sčítáním, násobením, číslicemi a vnořenými závorkovanými strukturami.

Řešený příklad 29:

Gramatiku v Backusově-Naurově formě pro náš zjednodušený příklad (pouze s číslicemi místo identifikátorů) lze zapsat takto:

<Výraz> ::= <Term> { + <Term>}
<Term> ::= <Faktor>{ * <Faktor>}
<Faktor> ::= 0|1|2|…|9|(<Výraz>)

Nyní se schématicky pokusíme ukázat (nejde o zcela hotový kód), jak bychom sestrojili SA metodou rekurzivního sestupu pro tuto gramatiku v jazyce Pascal. Tento kód, pak umožňuje nejen SA, ale i detekci možných chyb. Nejprve sestrojíme proceduru, která zapouzdřuje celou činnost SA. Její hlavička může vypadat například takto:

```
procedure SyntaktickaAnalyza(infix:string;var err,pos:word);
      {procedura analyzuje aritmetický výraz infix, err obsahuje číslo chyby, pos
obsahuje pozici ,kde analýza skončila}
```

Používají se proměnné infixpos (pozice aktuálně čteného znaku ze vstupu), ch (aktuální znak).

Analyzátor dále obsahuje nezbytný lexikální analyzátor pro načítaní jednotlivých symbolů (v našem zjednodušeném případě jde o jednoznakové symboly). Lexikální analýza je realizována procedurou GetChar, která ukládá znak do proměnné ch a případně provede detekci chybové situace err=2, pokud načteme zcela nepřípustný znak.

```
procedure Getchar;      {čte znak z infixu do proměnné ch}
begin
  if  err=0 then
    begin
      Inc(infixpos);
      if infixpos<=Length(infix) then ch:=infix[infixpos] else ch:=#0;
      ch:=Upcase(ch);
              if not((ch in cislice)or(ch in ['(',')','*','+'])) then err:=2;     {ošetření
          nežádoucích znaků}
    end;
end;
```

Jádrem analyzátoru jsou jednotlivé rekurzivní procedury Výraz (sčítaní), Term (násobení), Faktor (číslice, vnořený závorkovaný výraz). Výraz přesně podle BNF buď volá podřízený Faktor nebo čte terminální symboly.

```
procedure Vyraz;            {výraz s nižší prioritou}
begin
  if err=0 then
    begin
      Term;
      while (ch='+') do
        begin
              Getchar;              {sčítání}
              Term;
        end;
    end;
end;

procedure Term;             {výraz s vyšší prioritou}
begin
  if err=0 then
    begin
      Faktor;
      while (ch='*') do
        begin
              Getchar;              {násobení}
              Faktor;
        end;
    end;
end;
```

A dále musíme sestrojit proceduru pro Faktor, která bude mít vzhledem k jinému charakteru přepisovaného řetězce i jiný kód.

```
procedure Faktor;   {synt. analýza operandu}
begin
  if err=0 then
    begin
      case ch of
      '0'..'9':
         begin                          {anal. číslic}
           Getchar;
         end;
      '(':begin
           Getchar;
                                {analýza výrazu se závorkou}
           Vyraz;
           if (ch<>')')and(err=0) then err:=4   {chyba- není ukončen závorkou}
           else if err=0 then
             begin
               Getchar;
             end;
         end;
      else if err=0 then err:=5;    {nebyl detekován ani vyraz,ani číslice}
      end;
    end;
```

Faktor tedy rozlišuje dvě situace – buď jde o číslici nebo jde o výraz začínající závorkou a ukončený opačnou závorkou. Logicky tedy můžeme odhalit další dvě chyby (err=4, když chybí závorka, err=5, když není detekován ani výraz ani číslice).
Pozn. Samozřejmě, že chybové detekce by mohly odhalit ještě další problematické konstrukce – např. skončení nejvyššího volání procedury Výraz před přečtením posledního znaku apod.

Ilustrujme nyní průběh výpočtu procedury Výraz na výrazu 5 + 3 * 2.

Infixová notace	**Aktuální znak**	**Aktuální procedura**	**Návrat do procedury**
5 + 3 * 2	5	Výraz	
5 + 3 * 2	5	Term	
+ 3 * 2	+	Faktor	Term
+ 3 * 2	+	Term	Výraz
3 * 2	3	Výraz (+)	
3 * 2	3	Term	
* 2	*	Faktor	Term
2	2	Term (*)	
		Faktor	Term (*)
		Term (*)	Výraz (+)
		Vyraz (+)	

9.5. Jiné metody

Existují samozřejmě i jiné metody implementace syntaktických analyzátorů. Za zmínku stojí například metoda spojových seznamů, která

vychází z toho, že přepisovací pravidlo pro neterminál si lze představit jako seznam neterminálů a terminálů reprezentujících řetězec, na který pravidlo přepisuje. Jednotlivé přepisovací pravidla pro daný neterminál tedy lze implementovat jako spojový seznam, který obsahuje buď uzly terminální, kdy srovnáváme symboly s analyzovaným řetězcem nebo neterminály, které obsahují pouze odkaz (ukazatel) na jiný spojový seznam, který reprezentuje daný neterminál. Principiálně jde vlastně o analogii rekurzivního sestupu, ale u této implementace není nutná rekurzivita navzájem vnořených procedur a tedy i volání. Iterativnost analýzy namísto rekurzivity může být výhodná zvláště pro velmi složité gramatiky a také je její velká potenciální síla v možnosti měnit gramatiku za běhu programu. Jistě si dokážete představit „elastický“ informační systém, který by umožňoval dokonce vytvářet nové syntaktické struktury svých vstupů, případně programovací jazyk, který byste takto mohli interaktivně rozšiřovat o nové konstrukce bez nutnosti zasahovat přímo do zdrojového kódu jeho překladače!

Nejdůležitější probrané pojmy:
- analýza obecných BKG (Earleyho algoritmus, CYK algoritmus)
- metoda rekurzivního sestupu

Úkol k textu:

Pokuste se promyslet, jak byste u příkladu na metodu rekurzivního sestupu odhalili chybu typu „chybí operátor“. Dále se pokuste vytvořit spojové seznamy pro syntaktickou analýzu pro stejný jazyk jako u metody rekurzivního sestupu.

Korespondenční úkol:

Sestrojte BNF pro jazyk logických výrazů výrokové logiky, kde můžete používat konjukci, disjunkci, implikaci, negaci a dále symboly a..z pro výrokové proměnné, symboly 0 a 1 pro true a false a rovněž můžete do závorek vnořit výraz stejného typu.
Pro sestrojenou gramatiku vytvořte metodou rekurzivního sestupu program v libovolném strukturovaném programovacím jazyce (např. Pascal), který bude provádět syntaktickou analýzu libovolné formule a jejich chybovou detekci.

10. Modelové aplikace pro syntaktickou analýzu

Cíl:

Tato kapitola Vám stručně přiblíží nástroje, které máte k dispozici k vlastnímu zkoumání syntaktických analyzátorů na počítači.

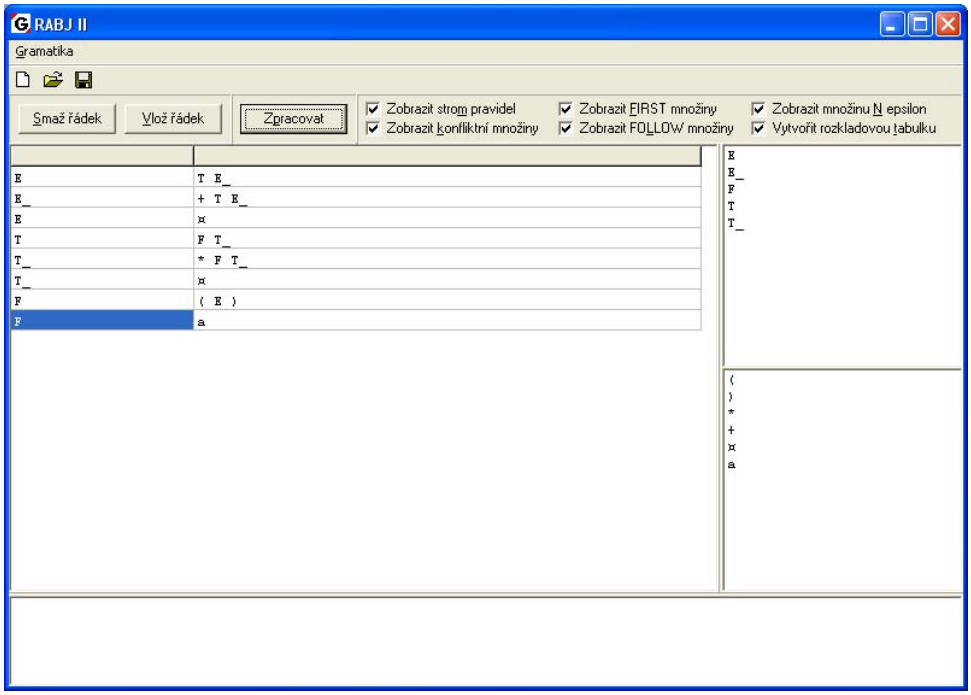
Postupy, které jsme se naučili v této studijní opoře, jsou relativně jednoduše implementovatelné. Ukážeme si dvě existující implementace vzniklé jako práce studentů Ostravské Univerzity. Jejich spustitelné soubory a zdrojové kódy jsou k dispozici jako příloha této práce.

10.1. Aplikace pro tvorbu rozkladových tabulek

První prací je aplikace RABJ II studenta pana Aleše Koběrského na tvorbu rozkladových tabulek pro LL(1) gramatiky. Umožňuje zadávat obecnou BKG gramatiku ve formě přepisovacích pravidel (symbol ¤ reprezentuje prázdný řetězec). Následně umožní tyto operace:

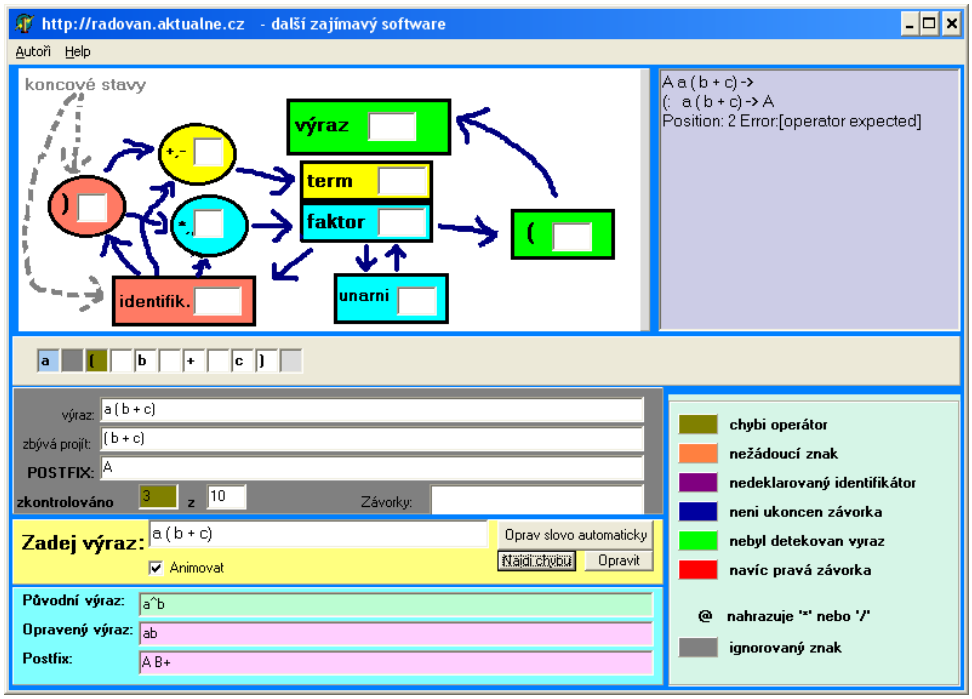
- zobrazení množiny pravidel
- výpočet množin FIRST a FOLLOW
- znázornění konfliktů z hlediska podmínek LL(1) gramatiky
- v případě, že není žádný konflikt, sestrojení rozkladové tabulky

Aplikace již obsahuje vytvořené příklady včetně analogických gramatik jaké byly řešeny v tomto textu.



10.2. Demonstrace metody rekurzivního sestupu

Další prací je animace metody rekurzivního sestupu, které vytvořili studenti Radovan Kaluža a Martin Svitánek. Aplikace umožňuje interaktivně provádět simulaci principu volání procedur při rekurzivním sestupu. Lze zadat aritmetický výraz s identifikátory a provést jeho syntaktickou analýzu s detekcí chyb.



Literatura

[Ce92] ČEŠKA, Milan, RÁBOVÁ, Zdena. Gramatiky a jazyky. Brno, VUT 1992.Dokument dostupný na URL (2005): http://www.fit.vutbr.cz/study/courses/TI1/public/Texty/ti.pdf

[Dv92] DVOŘAK, Stanislav. Dekompozice a rekurzivní algoritmy. Grada 1992, Praha

[Ch84] CHYTIL, Milan. Automaty a gramatiky. Praha, SNTL 1984.

[Ha03] HABIBALLA, H. Regulární a bezkontextové jazyky I.. Ostrava : Ostravská Univerzita, 2003. 140 s.

[Ho79] HOPCROFT, J.E., ULLMAN, J. D. Introduction to Automata theory, Languages and Computation. Addison-Wesley, Reading (Mass.), 1979

[Ka02] KASTENS, U.: Demonstration of parsing methods, http://www.uni-paderborn.de\fachbereich\AG\agkastens\compiler\parsdemo\index.html

[Le02] LEWIS, F.D. Recursive Descent Parsing, http://cs.engr.uky.edu\~lewis\essays\compilers\rec-des.html

[Ja97] JANČAR, Petr. Teorie jazyků a automatů. VŠB TU Ostrava, Dokument dostupný na URL: http://www.cs.vsb.cz/jancar/ (2005)

[Ja97a] JANČAR, Petr. Vyčíslitelnost a složitost. VŠB TU Ostrava, Dokument dostupný na URL: http://www.cs.vsb.cz/jancar/ (2005)

[Ji88] JINOCH, J., MULLER, K., VOGEL, J. Programování v jazyce Pascal. SNTL 1988, Praha

[Pa02] PAVLISKA, Viktor: Vyčíslitelnost a složitost I. distanční studijní text OU, 2002

[Wi05] Wikipedia: The free encyklopedia, dokument dostupný na: http://www.wikipedia.org (2005)

Na Internetu lze najít množství odkazů a materiálů – především v angličtině, nicméně existují i české a slovenské webovské stránky s materiály: např. http://www.fimuni.org/ apod. (2005)