VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GRAMATICKÉ SYSTÉMY A SYNTAXÍ ŘÍZENÝ PŘEKLAD ZALOŽENÝ NA NICH

GRAMMAR SYSTEMS AND SYNTAX-DIRECTED TRANSLATION BASED ON THEM

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE — Bc. JAROSLAV HANDLÍŘ
AUTHOR

VEDOUCÍ PRÁCE — Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2017

Abstrakt

Práce zkoumá teorii formálních jazyků v oblasti bezkontextových gramatik. Zaměřuje se především na možnosti a modely spolupráce více gramatik při řešení společného problému. V těchto souvislostech představuje gramatické systémy, které byly navrženy jako formální prostředek pro popis distribuovaného a paralelního zpracování. Po uvedení do dané problematiky se práce zaměřuje na praktické uplatnění těchto mechanismů při syntaxí řízeném překladu, a proto je druhá část práce věnována implementaci dynamického syntaktického analyzátoru, který během analýzy aplikuje více gramatik. S ohledem na co největší uživatelskou přívětivost a možné didaktické použití je aplikace implementována pomocí moderních webových technologií HTML5, JavaScript, AngularJS, CSS3, LESS a další.

Abstract

The thesis examines the theory of formal languages in the field of context-free grammars. It focuses mainly on the possibilities and models of collaborating grammars to solve a common problem. In this context, it presents grammatical systems that have been designed as a formal means for describing distributed and parallel processing. After introducing to the problematics, the thesis focuses on the practical use of these mechanisms in the translation controlled syntax, and therefore the second part of the thesis deals with the implementation of a dynamic syntactic analyzer that uses more grammars during the analysis. With respect to the greatest user friendliness and the possible didactic use, the application is implemented using modern web technologies HTML5, JavaScript, AngularJS, CSS3, LESS and more.

Klíčová slova

bezkontextové gramatiky, gramatické systémy, CD, PC, syntaktická analýza, precedenční analýza, syntaxí řízený překlad, dynamická gramatika, HTML5, JavaScript, AngularJS, CSS3, LESS

Keywords

context-free grammars, grammar systems, CD, PC, syntax analysis, precedence analysis, syntax-directed translation, dynamic grammar, HTML5, JavaScript, AngularJS, CSS3, LESS

Citace

HANDLÍŘ, Jaroslav. *Gramatické systémy a syntaxí řízený překlad založený na nich*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexander Meduna, CSc.

Gramatické systémy a syntaxí řízený překlad založený na nich

Prohlášení

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

……………………
Jaroslav Handlíř
15. května 2017

Poděkování

Tímto bych rád poděkoval svému vedoucímu diplomové práce prof. RNDr. Alexanderu Medunovi, CSc., za jeho čas a zájem, odborný dohled a vedení v průběhu celé práce.

Obsah

1 Úvod ... 2
2 Základní formální prostředky ... 3
3 Regulární gramatiky ... 4
3.1 Konečné automaty ... 5
4 Bezkontextové gramatiky ... 6
4.1 Derivační stromy a nejednoznačnost ... 7
4.2 Zásobníkové automaty ... 9
5 Gramatické systémy ... 10
5.1 CD gramatické systémy ... 11
5.1.1 Formální definice ... 11
5.1.2 Derivační módy ... 12
5.1.3 Generovaný jazyk ... 13
5.1.4 Příklady ... 13
5.1.5 Třídy jazyků CD gramatických systémů a jejich generativní síla ... 15
5.1.6 Hybridní CD gramatické systémy ... 16
5.1.7 Třídy jazyků hybridních CD gramatických systémů a jejich generativní síla ... 17
5.2 PC gramatické systémy ... 18
5.2.1 Formální definice ... 18
5.2.2 Derivační kroky ... 18
5.2.3 Generovaný jazyk ... 20
5.2.4 Varianty PC gramatických systémů ... 20
5.2.5 Příklad ... 21
5.2.6 Třídy jazyků PC gramatických systémů a jejich generativní síla ... 22
6 Syntaxí řízený překlad ... 23
6.1 LL gramatiky bez epsilon pravidel ... 26
6.1.1 Množina First ... 27
6.2 LL gramatiky s epsilon pravidly ... 29
6.2.1 Množina Empty ... 30
6.2.2 Množina First ... 32
6.2.3 Algoritmus First($X_1X_2...X_n$) ... 34
6.2.4 Algoritmus Empty($X_1X_2...X_n$) ... 34
6.2.5 Množina Follow ... 35
6.2.6 Množina Predict ... 36
6.3 Rekurzivní sestup ... 38
6.4 Prediktivní syntaktická analýza ... 39
6.5 LR syntaktický analyzátor ... 40
6.6 Precedenční syntaktický analyzátor ... 41
7 Aplikace ... 43
7.1 Uživatelské rozhraní ... 47
7.2 Implementace ... 48
7.3 Nasazení a vývoj ... 49
8 Závěr ... 50

1 Úvod

Práce se věnuje teorii formálních jazyků v oblasti bezkontextových gramatik, které jsou velmi významné především při analýze a překladu programovacích jazyků. Umožňují tak nejen formálně popsat a specifikovat daný programovací jazyk, ale zároveň provádět jeho syntaktickou analýzu a především pak transformaci většinou velmi abstraktního jazyka, který je však pro člověka dobře čitelný, do jazyka symbolických adres nebo přímo strojového kódu, aby mohl být daný program vykonávaný počítačem. Přestože lze popsat programovací jazyk jednou komplexní gramatikou, může být v mnoha ohledech užitečné zapojit do činnosti více gramatik, z nichž každá dokáže zpracovat např. pouze část jazyka nebo dokonce může být vstupní program tvořený více různými jazyky. Jedná se o spolupráci více gramatik při řešení společného problému. V běžném světě se tato situace vyskytuje denně. Tým lidí se podílí na jednom projektu, který se snaží společně a co nejefektivněji dokončit. V praxi se tyto přístupy často uplatňují i v oblasti překladu jazyků, ale zpravidla se klade důraz na funkčnost a dále se již nezkoumá teoretické pozadí a nehledají se formalismy. Avšak formální prostředky, které se snaží tuto problematiku zachytit a rozvinout, již dlouhou dobu existují. Řeč je o tzv. gramatických systémech, kterým je věnováno jádro této práce.

Gramatické systémy vznikly ve druhé polovině 20. století a definují různé modely spolupracujících gramatik, které v době jejich vzniku nemohli autoři z dnešního pohledu plně docenit. Tyto systémy zavádějí dva hlavní typy, které umožňují popis distribuovaného a paralelního zpracování. První typ se nazývá kooperující distribuované gramatické systémy, zkráceně CD gramatické systémy (z anglického *Cooperating Distributed*), a druhý typ se jmenuje paralelně komunikující gramatické systémy, zkráceně PC gramatické systémy (z anglického *Parallel Communicating*).

CD gramatické systémy jsou distribuovanou variantou a jednotlivé gramatiky systému (nazývány komponenty systému) pracují sekvenčně. Postupně se střídají při zpracování větné formy, kterou společně sdílí, dokud nenaleznou její řešení. Výsledek je tak dán dílčím přispěním všech gramatik, které mezi sebou přepínají podle různých pravidel. Obecnou analogii k tomuto systému je možné v literatuře nalézt např. u tzv. „problému tabule" (anglicky *The Black Board Model*), kdy se více lidí střídá u tabule. Společně tak řeší daný problém a každý z účastníků přispěje svými nápady. Za tímto systémem se zároveň skrývá i velký potenciál distribuce výkonu, který v době vzniku těchto systémů nebyl tolik využitelný. Naopak dnes, kdy je internet samozřejmostí, většina služeb se přesunuje na cloud a data i výkon lze snadno distribuovat díky moderním sítím po celém světě, je myšlenka CD gramatického systému velmi aktuální.

PC gramatické systémy pracují podobným způsobem, ale na rozdíl od předchozí sekvenční distribuované varianty jsou plně paralelní. Jednotlivé komponenty systému (gramatiky) pracují paralelně a každá komponenta zpracovává svoji vlastní větnou formu. Mezi sebou mohou komunikovat a předávat si informace pomocí tzv. komunikačních symbolů. Tyto systémy tak dávají prostor nejen pro řízení výpočtu (může existovat hlavní komponenta, která si žádá práci ostatních, jedná se potom o centralizovaný režim), ale především tak vznikl formální prostředek pro zavedení paralelismu. Paralelní zpracování je z dnešního pohledu druhým velmi významným trendem, protože dnes neexistuje obyčejný chytrý telefon, který by nedisponoval vícejádrovým procesorem. Otevírá se tím další velká oblast využití těchto systémů.

První polovina práce pojednává o obecných základech formálních jazyků a bezkontextových gramatik, které jsou aplikovány v gramatických systémech. Poté již následuje podrobné

vysvětlení obou typů gramatických systémů doplněné jejich obecnými definicemi, variantami a možnostmi spolu s praktickými příklady ukazujícími způsob jejich použití.

Druhá část práce se věnuje využití gramatických systémů při syntaktické analýze. Jednotlivé kapitoly popisují různé možnosti přístupu k syntaktické analýze a syntaxí řízenému překladu, přičemž podrobněji jsou popsány ty metody, které byly použity v související praktické práci.

Praktická část práce kombinuje přístupy syntaktické analýzy shora dolů s analýzou zdola nahoru pomocí centralizovaného PC gramatického systému do moderní webové aplikace, která uživateli umožňuje nejen velmi jednoduchým způsobem provést syntaktickou analýzu svého vlastního vstupního kódu v prostředí internetového prohlížeče bez nutnosti cokoli instalovat a konfigurovat, ale zároveň dává možnost si přímo v prohlížeči definovat velmi snadno vlastní gramatiku, která bude následně použita při analýze. Výstup syntaktické analýzy informuje uživatele o tom, jestli analýza proběhla v pořádku, příp. kde se pravděpodobně ve vstupním textu nachází chyba. Dále se uživatel dozví posloupnost aplikovaných pravidel gramatiky spolu s odpovídající LL tabulkou. Celá aplikace má tedy především didaktický charakter, který by měl uživateli pomoci k lepšímu pochopení bezkontextových gramatik, syntaktické analýzy a gramatických systémů. Zároveň aplikace představuje použití moderních vývojářských přístupů a webových technologií v praxi. Pro implementaci byly použity HTML5, kaskádové styly CSS3 a LESS, klientský JavaScript s frameworkem AngularJS, pro responzitivitu a další styly framework Bootstrap. Tímto však výčet použitých technologií a nástrojů nekončí a jejich podrobný popis lze nalézt v kapitole 7. Za zmínku také stojí sekce příloh, která obsahuje ukázky reálné aplikace a jejího použití.

2 Základní formální prostředky

Tato kapitola seznamuje čtenáře se základními prostředky formálních jazyků, které jsou použity v celé následující práci, jež si klade za cíl nejen prohloubení znalostí, ale i srozumitelné vysvětlení základních principů pro čtenáře, kteří zatím spíše tápou. Zkušenější čtenáři mohou tedy této kapitole věnovat menší pozornost.

Abeceda je konečná, neprázdná množina elementů, které se nazývají symboly. Značí se $\boldsymbol{\Sigma}$[1]. Abecedu, která obsahuje tři symboly $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$, lze zapsat následovně:

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

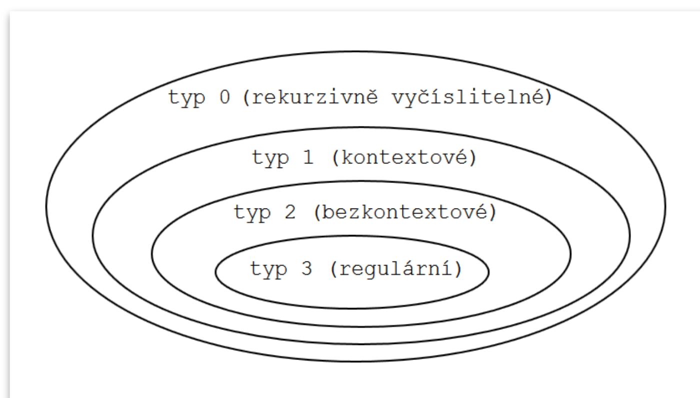
Řetězec je potom posloupnost symbolů dané abecedy, která může mít nulovou délku. Takový řetězec se značí $\boldsymbol{\varepsilon}$[2] a platí, že $\boldsymbol{\varepsilon}$ je řetězec nad abecedou $\boldsymbol{\Sigma}$. $\boldsymbol{\varepsilon}$ je tedy řetězec nulové délky, tzn. prázdný řetězec. Jinak platí, že pokud je $\boldsymbol{x}$ řetězec nad $\boldsymbol{\Sigma}$ a $\boldsymbol{a} \in \boldsymbol{\Sigma}$, potom i $\boldsymbol{xa}$ je řetězec nad abecedou $\boldsymbol{\Sigma}$.

Jazykem se nazývá každá podmnožina řetězců nad danou abecedou, tedy nechť $\boldsymbol{\Sigma}^*$ značí množinu všech řetězců nad abecedou $\boldsymbol{\Sigma}$, potom každá podmnožina $\boldsymbol{L} \subseteq \boldsymbol{\Sigma}^*$ je jazyk nad abecedou $\boldsymbol{\Sigma}$.

Chomského hierarchie definuje kaskádu jednotlivých rodin jazyků, kde nadřazený typ má větší vyjadřovací sílu, a rozšiřuje tak typ pod ním. Bezkontextové gramatiky jsou rodinou jazyků typu 2 a rozšiřují tak nejjednodušší typ jazyků, kterým jsou regulární jazyky. Regulární jazyky jsou samotným základem a ve většině publikací jimi začíná výklad. Popisují se na nich nejzákladnější principy

[1] řecké velké písmeno sigma
[2] řecké malé písmeno epsilon

operací nad formálními jazyky. Tato práce je sice věnována bezkontextovým gramatikám, ale přesto následující kapitola obsáhne stručný úvod do regulárních jazyků a gramatik. Regulární jazyky a především konečné automaty jsou jedním z jejich formálních prostředků a budou zapotřebí v praktické části. Podrobné informace lze nalézt např. v publikaci [2].



Obrázek 1: Chomského hierarchie jazyků

3 Regulární gramatiky

Úvodem je třeba ještě vysvětlit, co je to ***terminál*** a ***neterminál***. Za ***terminál*** či terminální symbol se označuje znak nebo řetězec z dané abecedy, zatímco ***neterminál*** představuje ostatní znaky, řetězce a symboly použité při definování pravidel gramatiky, čemuž je blíže věnována následující kapitola.

Regulární jazyky jsou nejjednodušší třídou jazyků v Chomského hierarchii a označují se typem 3. Obsahují všechny konečné jazyky, tj. jazyky, jejichž řetězce mají konečnou délku, a jednodušší případy nekonečných jazyků. Jednoduchým příkladem může být jazyk nad abecedou $\boldsymbol{\Sigma} = \{\boldsymbol{a}, \boldsymbol{b}\}$, který obsahuje nejdříve sekvenci symbolů $\boldsymbol{a}$ a poté sekvenci symbolů $\boldsymbol{b}$. Formální zápis takového jazyka by vypadal následovně:

$$\boldsymbol{L = \{a^n b^m : n \geq 0, m \geq 0\}}$$

Jedním z formalismů regulárních jazyků jsou regulární výrazy. Jedná se o sadu pravidel a operací, které umožňují popsat regulární jazyky. Přestože regulární výrazy v rámci této práce přímo použity nejsou, jejich principy se prolínají i do bezkontextových gramatik, a je proto vhodné si je alespoň ve stručnosti definovat.

Rozlišují se tři základní operace nad řetězci, potažmo jazyky, a to

- ***konkatenace***, značí se operátorem . a provádí zřetězení dvou řetězců,
- ***sjednocení***, značí se $+$ a umožňuje výběr,
- ***iterace***, která se značí $*$,

a jejich vzájemná priorita je $* > . > +$.

Regulární výrazy nad abecedou $\boldsymbol{\Sigma}$ a jazyky, které značí, lze definovat následovně:

- $\emptyset$ je regulární výraz značící prázdnou množinu (prázdný jazyk)

- $\boldsymbol{\varepsilon}$ je regulární výraz značící jazyk $\{\boldsymbol{\varepsilon}\}$
- $\boldsymbol{a}$, kde $\boldsymbol{a} \in \boldsymbol{\Sigma}$, je regulární výraz značící jazyk $\{\boldsymbol{a}\}$
- Nechť $\boldsymbol{r}$ a $\mathbf{s}$ jsou regulární výrazy značící jazyky $\boldsymbol{L_r}$ a $\boldsymbol{L_s}$, potom:
    - $(\boldsymbol{r}.\boldsymbol{s})$ je regulární výraz značící jazyk $\boldsymbol{L} = \boldsymbol{L_r}\boldsymbol{L_s}$
    - $(\boldsymbol{r} + \boldsymbol{s})$ je regulární výraz značící jazyk $\boldsymbol{L} = \boldsymbol{L_r} \cup \boldsymbol{L_s}$
    - $(\boldsymbol{r}^*)$ je regulární výraz značící jazyk $\boldsymbol{L} = \boldsymbol{L_r}^*$

Regulární jazyk, jehož příklad je uveden na začátku této kapitoly, lze definovat následovně:

> Nechť $\boldsymbol{L}$ je jazyk, potom je $\boldsymbol{L}$ regulární jazyk, pokud existuje regulární výraz $\boldsymbol{r}$, který tento jazyk značí.

Neformálně řečeno, jazyk $\boldsymbol{L}$ je jazykem regulárním, pokud je možno pro tento jazyk najít regulární výraz, který jej popisuje. Dalšími možnostmi, jak specifikovat regulární jazyky, jsou regulární gramatiky nebo konečné automaty, jejichž definice zakončí tuto doplňující kapitolu.

Regulární gramatika $\boldsymbol{G}$ je čtveřice $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$, kde:

- $\boldsymbol{N}$ je abeceda neterminálů
- $\boldsymbol{T}$ je abeceda terminálů a platí, že $\boldsymbol{N} \cap \boldsymbol{T} = \emptyset$
- $\boldsymbol{P}$ je konečná množina pravidel tvaru:
    - $\boldsymbol{A} \to \boldsymbol{xB}$, kde $\boldsymbol{A}, \boldsymbol{B} \in \boldsymbol{N}, \boldsymbol{x} \in \boldsymbol{T}$ nebo
    - $\boldsymbol{A} \to \boldsymbol{x}$, kde $\boldsymbol{A} \in \boldsymbol{N}, \boldsymbol{x} \in \boldsymbol{T}$ případně
    - $\boldsymbol{S} \to \boldsymbol{\varepsilon}$, pokud se $\boldsymbol{S}$ neobjevuje na pravé straně žádného pravidla
- $\boldsymbol{S} \in \boldsymbol{N}$ je počáteční (startující) neterminál

3.1 Konečné automaty

Konečný automat je velmi užitečný nástroj, který umožňuje nejen přímou implementaci lexikální analýzy, o které bude zmínka později, ale využívá se v celé řadě aplikací praktických problémů, kde se systém nachází v určitém stavu a za daných podmínek přechází do stavu následujícího. Mimo to se velmi často používá grafická reprezentace pro názorné definování daného regulárního jazyka nebo obecně modelovaného problému.

Konečný automat je pětice $\boldsymbol{M} = (\boldsymbol{Q}, \boldsymbol{\Sigma}, \boldsymbol{R}, \boldsymbol{s}, \boldsymbol{F})$, kde:

- $\boldsymbol{Q}$ je konečná neprázdná množina stavů
- $\boldsymbol{\Sigma}$ je vstupní abeceda
- $\boldsymbol{R}$ je konečná množina pravidel tvaru $\boldsymbol{pa} \to \boldsymbol{q}$, kde $\boldsymbol{p}, \boldsymbol{q} \in \boldsymbol{Q}, \boldsymbol{a} \in \boldsymbol{\Sigma} \cup \{\boldsymbol{\varepsilon}\}$
- $\boldsymbol{s} \in \boldsymbol{Q}$ je počáteční stav
- $\boldsymbol{F} \subseteq \boldsymbol{Q}$ je množina koncových stavů

Konečný automat pracuje tak, že má k dispozici sadu stavů, ve kterých se může nacházet, a čtecí hlavu, pomocí níž čte symboly vstupní pásky po jednom zleva doprava. Na začátku je inicializován na počáteční stav $\boldsymbol{S}$ a čtecí hlava se nachází na první pozici pásky, tedy prvním symbolu. Poté postupně čte symboly z pásky a pro každý přečtený symbol a aktuální stav vyhledá v množině

pravidel, do jakého následujícího stavu může přejít přečtením tohoto symbolu. Pokud přečetl všechny symboly a dostal se do jednoho z koncových stavů $\boldsymbol{f}$, pak může daný jazyk přijmout a tento jazyk je regulární.

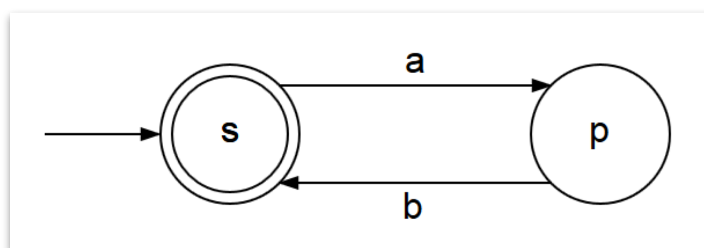
Formální definice jazyka přijímaného konečným automatem je následující. Nechť $\boldsymbol{M} = (\boldsymbol{Q}, \boldsymbol{\Sigma}, \boldsymbol{R}, \boldsymbol{s}, \boldsymbol{F})$ je konečný automat. Jazyk přijímaný konečným automatem $\boldsymbol{M}$, $\boldsymbol{L}(\boldsymbol{M})$, je definován:

$$\boldsymbol{L}(\boldsymbol{M}) = \{\boldsymbol{w}: \boldsymbol{w} \in \boldsymbol{\Sigma}^*, \boldsymbol{sw} \vdash^* \boldsymbol{f}, \boldsymbol{f} \in \boldsymbol{F}\}$$

Příklad 1.: Mějme konečný automat $\boldsymbol{M} = (\{\boldsymbol{s}, \boldsymbol{q}\}, \{\boldsymbol{a}, \boldsymbol{b}\}, \boldsymbol{R}, \boldsymbol{s}, \{\boldsymbol{s}\})$, kde R obsahuje pravidla:

- $\boldsymbol{sa} \longrightarrow \boldsymbol{q}$
- $\boldsymbol{qb} \longrightarrow \boldsymbol{s}$

Jazyk přijímaný tímto automatem je $\boldsymbol{L}(\boldsymbol{M}) = \{(\boldsymbol{ab})^{\boldsymbol{n}} : \boldsymbol{n} \geq \boldsymbol{0}\}$ a grafická reprezentace vypadá následovně:



Obrázek 2: Konečný automat reprezentující jazyk (ab)*

4 Bezkontextové gramatiky

Následující kapitoly vychází z poznatků předchozích sekcí a postupně staví na tomto základu, takže nebude již vše vysvětlováno tak podrobně, jako tomu bylo v kapitole 2 a 3.

Bezkontextové jazyky, jak již název napovídá, nejsou omezovány vnitřními vazbami nebo vztahy na základě obsahu, tj. kontextu, a jediné, co určuje syntaktickou strukturu bezkontextového jazyka, jsou pravidla gramatiky.

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

- $\boldsymbol{N}$ je abeceda neterminálů
- $\boldsymbol{T}$ je abeceda terminálů a platí, že $\boldsymbol{N} \cap \boldsymbol{T} = \emptyset$
- $\boldsymbol{P}$ je konečná množina pravidel tvaru $\boldsymbol{A} \rightarrow \boldsymbol{x}$, kde $\boldsymbol{A} \in \boldsymbol{N}, \boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T})^*$
- $\boldsymbol{S} \in \boldsymbol{N}$ je počáteční (startující) neterminál

Derivačním krokem se označuje akce, kdy se v dané větné formě provede změna řetězce za použití některého z pravidel v množině $\boldsymbol{P}$. Obecná definice potom zní následovně. Nechť $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ je bezkontextovou gramatikou, $\boldsymbol{u}, \boldsymbol{v} \in (\boldsymbol{N} \cup \boldsymbol{T})^*$, tedy $\boldsymbol{u}, \boldsymbol{v}$ jsou řetězce terminálních a neterminálních symbolů, které mohou být i prázdné, a $\boldsymbol{p} = \boldsymbol{A} \rightarrow \boldsymbol{x} \in \boldsymbol{P}$. Potom $\boldsymbol{uAv}$ přímo derivuje $\boldsymbol{uxv}$ za použití pravidla $\boldsymbol{p}$ v gramatice $\boldsymbol{G}$, což lze formálně zapsat $\boldsymbol{uAv} \Rightarrow \boldsymbol{uxv}\ [\boldsymbol{p}]$ či jednoduše $\boldsymbol{uAv} \Rightarrow \boldsymbol{uxv}$. Tato

definice neříká nic jiného, než že je možné přepsat neterminál $\boldsymbol{A}$ ve větě $\boldsymbol{uAv}$ řetězcem terminálů a neterminálů, popř. prázdným řetězcem (došlo by ke smazání neterminálu $\boldsymbol{A}$), pokud v gramatice existuje příslušné pravidlo. Aplikováním více derivačních kroků vzniká sekvence derivačních kroků, ale její definice zde již nebude zmíněna, bližší podrobnosti lze nalézt např. v publikaci [2].

Jazyk generovaný bezkontextovou gramatikou je takový, jehož všechny řetězce $\boldsymbol{w}$ lze vygenerovat pomocí sekvence derivačních kroků z počátečního neterminálu $\boldsymbol{S}$. Formální zápis vypadá následovně. Nechť $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ je bezkontextová gramatika, pak jazyk generovaný bezkontextovou gramatikou $\boldsymbol{G}$, $\boldsymbol{L}(\boldsymbol{G})$, je definován:

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

Postupně tedy dochází k expanzi a přepisování neterminálů za pomoci pravidel gramatiky počínaje startujícím neterminálem $\boldsymbol{S}$, až vznikne výsledný řetězec terminálů.

4.1 Derivační stromy a nejednoznačnost

Derivační strom je grafická reprezentace jedné či více derivací aplikovaných na danou větnou formu v podobě orientovaného grafu. Jeho nespornou výhodou je jasné a přehledné zobrazení provedených derivačních kroků. Mimo to umožňuje lépe pochopit některé problémy, které vyplývají z obecnosti bezkontextových gramatik a které by nemusely být hned zřejmé, obzvláště bez grafického znázornění.

Řeč je o nejednoznačnosti bezkontextových gramatik. Obecně bezkontextové gramatiky umožňují provést v každém derivačním kroku přepsání kteréhokoliv neterminálu za použití libovolného pravidla z množiny pravidel, které lze v daném kroku aplikovat na zpracovávanou větnou formu. V teoretickém světě by to nevadilo, naopak díky tomu vznikne v mnoha případech přehlednější gramatika, ale z hlediska praxe je to problém, protože různé posloupnosti aplikovaných pravidel na jedné a té samé větné formě mohou vygenerovat různé derivační stromy, což vede na již zmíněnou nejednoznačnost, a tedy i nedeterminističnost. V praxi je však potřeba mít jasně dané kroky, protože počítač nedokáže pracovat nedeterministicky, tedy hádat, co by bylo nejvhodnější. Určitě existují speciální případy a experimentální systémy, které se o to snaží, ale pro běžnou aplikaci syntaktických analyzátorů a překladačů je vyžadován deterministický přístup. Následující dva příklady se pokusí demonstrovat problém nejednoznačnosti na jednoduché gramatice pro základní matematické operace.

Příklad 2.: Nechť $\boldsymbol{G}$ je gramatika $\boldsymbol{G} = (\{\boldsymbol{E}\}, \{+, *, \boldsymbol{i}\}, \boldsymbol{P}, \boldsymbol{E})$, kde $\boldsymbol{P}$ obsahuje následující pravidla:

- $\boldsymbol{E} \rightarrow \boldsymbol{E} * \boldsymbol{E}$
- $\boldsymbol{E} \rightarrow \boldsymbol{E} + \boldsymbol{E}$
- $\boldsymbol{E} \rightarrow \boldsymbol{i}$

Vygenerujte řetězec $\boldsymbol{i} + \boldsymbol{i} * \boldsymbol{i}$.

Jak již bylo zmíněno výše, bezkontextové gramatiky neříkají, které neterminály se mají rozgenerovat v jakém pořadí, pokud jich větná forma obsahuje více. Generování začne tedy ze startujícího neterminálu a poté se vybírají neterminály v různém pořadí. Toto je první polovina problému nejednoznačnosti, která je ale snadno řešitelná. Lze zavést omezení, že se v každém derivačním kroku

zpracuje vždy určitý neterminál. V praxi se tak zavádí tzv. nejlevější, resp. nejpravější derivace, která v každém derivačním kroku rozgeneruje vždy nejlevější, resp. nejpravější neterminál. Protože se jedná o speciální případ obecné derivace, lze bez újmy na obecnosti prohlásit, že aplikování vždy nejlevější nebo nejpravější derivace nemá vliv na výsledek, ale zavede to jistý řád a jednotnost při zpracování. Podrobnější vysvětlení a důkazy lze nalézt v publikaci [2].

Náhodný výběr neterminálu

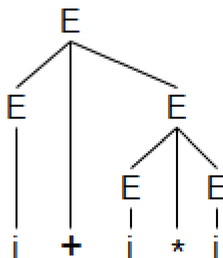
$$\begin{aligned} \underline{E} &\Rightarrow E + \underline{E} \\ &\Rightarrow E + \underline{E} * E \\ &\Rightarrow \underline{E} + i * E \\ &\Rightarrow i + i * \underline{E} \\ &\Rightarrow i + i * i \end{aligned}$$

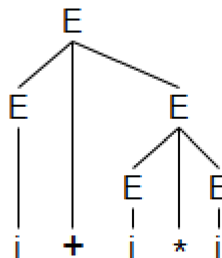
Nejlevější neterminál

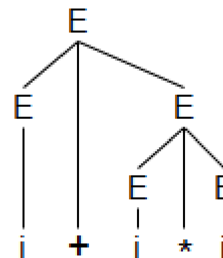
$$\begin{aligned} \underline{E} &\Rightarrow \underline{E} + E \\ &\Rightarrow \underline{E} + E * E \\ &\Rightarrow i + \underline{E} * E \\ &\Rightarrow i + i * \underline{E} \\ &\Rightarrow i + i * i \end{aligned}$$

Nejpravější neterminál

$$\begin{aligned} \underline{E} &\Rightarrow E + \underline{E} \\ &\Rightarrow E + E * \underline{E} \\ &\Rightarrow E + \underline{E} * i \\ &\Rightarrow \underline{E} + i * i \\ &\Rightarrow i + i * i \end{aligned}$$







I když se posloupnost provedených derivací liší, tak výsledný derivační strom je pro všechna pořadí (omezení) stejný. Tento výsledek je uspokojující, protože se dá použít nejlevější nebo nejpravější derivace beze změny výsledku. Bohužel ale, jak lze vidět vzápětí, problém nejednoznačnosti spočívá v tom, že gramatika dovoluje použít libovolné pravidlo z množiny aktuálně dostupných pravidel. Pokud se zahájí derivace pravidlem s operací krát místo plus (gramatika tento krok dovolí, jedná se o naprosto legitimní derivaci), pak je možné vygenerovat stejný řetězec terminálů jako v předchozí variantě.

Náhodný výběr neterminálu

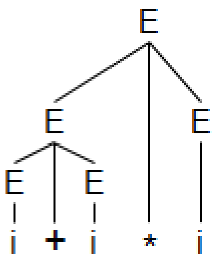
$$\begin{aligned} \underline{E} &\Rightarrow E * \underline{E} \\ &\Rightarrow \underline{E} * i \\ &\Rightarrow E + \underline{E} * i \\ &\Rightarrow \underline{E} + i * i \\ &\Rightarrow i + i * i \end{aligned}$$

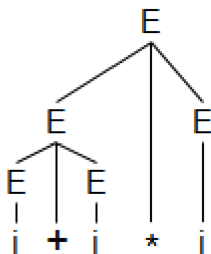
Nejlevější neterminál

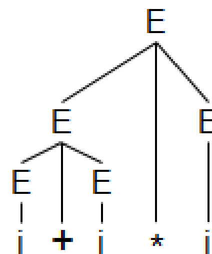
$$\begin{aligned} \underline{E} &\Rightarrow \underline{E} * E \\ &\Rightarrow \underline{E} + E * E \\ &\Rightarrow i + \underline{E} * E \\ &\Rightarrow i + i * \underline{E} \\ &\Rightarrow i + i * i \end{aligned}$$

Nejpravější neterminál

$$\begin{aligned} \underline{E} &\Rightarrow E * \underline{E} \\ &\Rightarrow \underline{E} * i \\ &\Rightarrow E + \underline{E} * i \\ &\Rightarrow \underline{E} + i * i \\ &\Rightarrow i + i * i \end{aligned}$$







Opět se neliší výsledný derivační strom pro různé pořadí zpracovaných neterminálů, ale to jen díky tomu, že se zachovala návaznost aplikovaných pravidel. Díky grafické reprezentaci, jak bylo zmíněno výše, je na první pohled patrné, že pro stejný řetězec terminálů existují dva různé derivační stromy, a tedy gramatika $\boldsymbol{G}$ je nejednoznačná.

Pro předejití nejednoznačnosti gramatiky, potažmo jazyka, je nutné zavést determinismus. K dalšímu výkladu jsou potřeba zásobníkové automaty, resp. deterministické zásobníkové automaty, kterým bude věnována následující podkapitola.

4.2 Zásobníkové automaty

Stejně jako u regulárních jazyků byl jedním z formálních aparátů konečný automat, tak u bezkontextových jazyků se jedná o zásobníkový automat. Principiálně se od konečného automatu příliš neliší. Opět má množinu stavů, ve kterých se může nacházet, a ze vstupní pásky čte symboly. Oproti konečnému automatu ale pracuje navíc se zásobníkem. Následující stav je tedy dán aktuálním stavem, symbolem čteným z pásky a symbolem na zásobníku. Podle těchto tří hodnot rozhoduje, které pravidlo z množiny pravidel aplikovat v aktuálním kroku. Formální definice zásobníkového automatu je následující:

Zásobníkový automat je sedmice $\boldsymbol{M = (Q, \Sigma, \Gamma, R, s, S, F)}$, kde:

- $\boldsymbol{Q}$ je konečná množina stavů
- $\boldsymbol{\Sigma}$ je vstupní abeceda
- $\boldsymbol{\Gamma}$ je zásobníková abeceda
- $\boldsymbol{R}$ je konečná množina pravidel tvaru $\boldsymbol{Apa \rightarrow wq}$, kde $\boldsymbol{A \in \Gamma,\ p, q \in Q, a \in \Sigma \cup \{\varepsilon\}}$, $\boldsymbol{w \in \Gamma^*}$
- $\boldsymbol{s \in Q}$ je počáteční stav
- $\boldsymbol{S \in \Gamma}$ je počáteční symbol na zásobníku
- $\boldsymbol{F \subseteq Q}$ je množina koncových stavů

Na rozdíl od konečného automatu zásobníkový automat může přijímat jazyk třemi způsoby, a to ***přechodem do koncového stavu*** (1), tento případ je totožný s konečným automatem, ***vyprázdněním zásobníku*** (2) nebo kombinací, tj. ***přechodem do koncového stavu a zároveň vyprázdněním zásobníku*** (3), přičemž všechny tyto varianty mají ekvivalentní vyjadřovací sílu. Formální definice jazyka přijímaného zásobníkovým automatem je následující. Nechť $\boldsymbol{M = (Q, \Sigma, \Gamma, R, s, S, F)}$ je zásobníkový automat. Jazyk přijímaný zásobníkovým automatem $\boldsymbol{M}$, $\boldsymbol{L(M)}$, je definován:

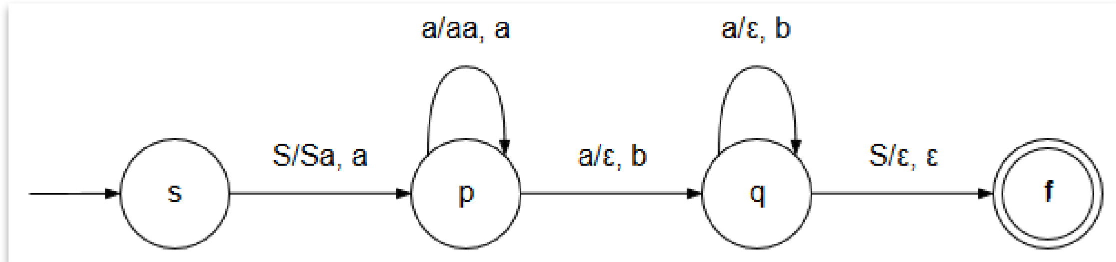
1. $\boldsymbol{L(M)_f = \{w: w \in \Sigma^*,\ Ssw \vdash^* zf,\ z \in \Gamma^*,\ f \in F\}}$
2. $\boldsymbol{L(M)_\varepsilon = \{w: w \in \Sigma^*,\ Ssw \vdash^* zf,\ z = \varepsilon,\ f \in Q\}}$
3. $\boldsymbol{L(M)_{f\varepsilon} = \{w: w \in \Sigma^*,\ Ssw \vdash^* zf,\ z = \varepsilon,\ f \in F\}}$

Příklad 1.: Mějme zásobníkový automat $\boldsymbol{M = (\{s, p, q, f\}, \{a, b\}, \{a, S\}, R, s, S, \{f\})}$, kde R obsahuje pravidla:

- $\boldsymbol{Ssa \longrightarrow Sap}$
- $\boldsymbol{apa \longrightarrow aap}$
- $\boldsymbol{apb \longrightarrow q}$

- $\boldsymbol{aqb \longrightarrow q}$
- $\boldsymbol{Sq \longrightarrow f}$

Jazyk přijímaný tímto automatem je $\boldsymbol{L(M)} = \{\boldsymbol{a^n b^n}: \boldsymbol{n} \geq \boldsymbol{1}\}$. Tento zásobníkový automat je navíc schopen přijmout všemi třemi způsoby, které jsou popsány výše, protože v okamžiku, kdy přejde do koncového stavu, má zároveň i prázdný zásobník. Grafická reprezentace vypadá následovně:



Obrázek 3: Zásobníkový automat reprezentující jazyk $a^n b^n,\ n \geq 1$

Deterministický zásobníkový automat je pro uplatnění bezkontextových gramatik z praktického hlediska velmi podstatný nástroj. Drobnou úpravou definice obecného zásobníkového automatu tak vznikne aparát, u kterého se zamezilo náhodnému výběru či rozhodování, tedy nedeterminismu, a opakovaný výpočet proběhne vždy stejným způsobem. Je tedy nutné zavést omezení, aby automat nemohl provést v každém kroku výpočtu více než jeden přechod. Tedy formálně musí platit, že pro každé pravidlo tvaru $\boldsymbol{Apa \longrightarrow wq} \in \boldsymbol{R}$ množina $\boldsymbol{R} - \{\boldsymbol{Apa \longrightarrow wq}\}$ neobsahuje žádné pravidlo s levou stranou $\boldsymbol{Apa}$ nebo $\boldsymbol{Ap}$.

Bohužel ne všechny bezkontextové gramaticky lze převést na ekvivalentní deterministickou formu, tedy třída ***deterministických bezkontextových jazyků*** ⊂ třída jazyků přijímaných ***obecnými (nedeterministickými) zásobníkovými automaty.***

5 Gramatické systémy

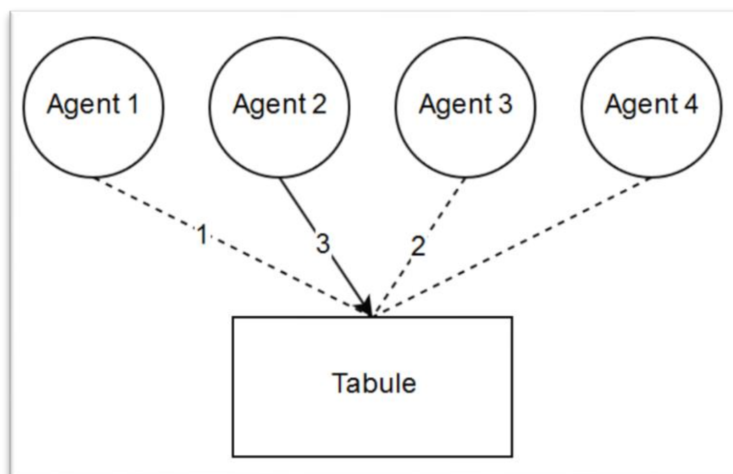
Gramatické systémy vznikly v průběhu druhé poloviny minulého století jako nová oblast formálních jazyků. Hlavní snahou bylo nalezení nástrojů a metod, které již v praxi existovaly dříve a běžně se používaly, ale nebyly pro ně vytvořeny formalismy a teoretický základ, o který by se dalo opřít při dalším zkoumání. V principu se gramatické systémy snaží pokrýt dvě oblasti informačních technologií, které nabraly velkého rozmachu právě ke konci minulého století a dnes je jejich vývoj stále velmi aktuální. Jedná se o distribuované a paralelní zpracování. V mnoha ohledech je výhodné i při běžné syntaktické analýze použít více gramatik, které se podílí na výpočtu. Pokud se k tomu zapojí možnosti distribuce výkonu nebo paralelního provádění, získá se nejen silný nástroje v rámci formálních jazyků, ale velké možnosti využití výpočetních zdrojů.

Nejlépe představitelnou variantou gramatického systému je syntaktická analýza, kdy se kombinuje přístup analýzy shora dolů s analýzou zdola nahoru. Dochází tak k přepínání dvou gramatik, které spolu komunikují a spolupracují při analýze, což umožňuje použít např. dvě jednodušší, přehlednější gramatiky, než se snažit celou analýzu provést pouze jednou gramatikou.

Následující dvě podkapitoly představí popořadě kooperující distribuované gramatické systémy, tzv. CD gramatické systémy, a paralelní gramatické systémy, tzv. PC gramatické systémy.

5.1 CD gramatické systémy

Gramatické systémy s označením CD definují model distribuované spolupráce gramatik, tedy kooperující distribuované gramatické systémy (z angličtiny *Cooperating Distributed Grammar Systems*). Tento systém si lze představit jako stolní hru. Hráči mají jasně dané pořadí a postupně se střídají. Když se hráč dostane na řadu, záleží na pravidlech hry, kdy musí svoji aktivitu předat dalšímu na řadě. Pravidla mohou být nastavena tak, že každý hráč provede přesně daný počet tahů (jeden nebo více) nebo provádí tahy tak dlouho, dokud může pokračovat. Poté předá hru dalšímu hráči, který je na řadě. Odborné články přirovnávají tento systém např. týmové spolupráci agentů, kteří řeší společný problém. Velmi často uváděný příklad je tzv. „problém tabule“, neboli *The Blackboard Model* či *The Blackboard System*. Skupina lidí řeší společný problém u tabule, kde se střídají, a každý z účastníků přispívá svými nápady, dokud dohromady problém nevyřeší.



Obrázek 4: Ukázka spolupráce agentů u tabule

CD gramatické systémy pracují sekvenčně na jedné sdílené větné formě. Každá gramatika, která je na řadě, provede modifikaci (příp. i více modifikací) větné formy a předá řízení další gramatice. Jednotlivé gramatiky jsou označovány jako tzv. komponenty systému. Následující podkapitoly uvedou jednotlivé definice a formalismy CD gramatických systémů.

5.1.1 Formální definice

V této práci jsou systémy definovány nad bezkontextovými gramatikami, které byly uvedeny v předchozí kapitole. CD gramatický systém $\boldsymbol{\Gamma}$ stupně $\boldsymbol{n}$, $\boldsymbol{n \geq 1}$, je $(n+3)$-tice, kde $\boldsymbol{n}$ značí počet množin pravidel systému, což lze chápat jako počet gramatik systému, které se budou podílet na výpočtu. Jednotlivé množiny pravidel (gramatiky) sdílí množiny terminálů, neterminálů a startující neterminál.

$$\boldsymbol{\Gamma} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{S}, \boldsymbol{P_1}, \dots, \boldsymbol{P_n})\text{, kde:}$$

- $\boldsymbol{N}$ – abeceda neterminálů
- $\boldsymbol{T}$ – abeceda terminálů, $\boldsymbol{N} \cap \boldsymbol{T} = \emptyset$
- $\boldsymbol{S}$ – startující neterminál, $\boldsymbol{S} \in \boldsymbol{N}$
- $\boldsymbol{P_1} \dots \boldsymbol{P_n}$ – konečná množina bezkontextových pravidel tvaru $\boldsymbol{a} \rightarrow \boldsymbol{x},\ \boldsymbol{a} \in \boldsymbol{N},\ \boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T})^*$ (tzv. komponenty systému)

Forlmálně lze zapsat tento systém i takto:

$$\boldsymbol{\Gamma} = \boldsymbol{G_1}, \dots, \boldsymbol{G_n}$$
$$\boldsymbol{G_i} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P_i}, \boldsymbol{S}) \text{ pro } \boldsymbol{1} \leq \boldsymbol{i} \leq \boldsymbol{n},$$

což vyplývá z popisu uvedeného výše. Zpravidla požadujeme, aby abecedy terminálů a neterminálů plus startující neterminál byly pro všechny participující gramatiky stejné, a ty se tak lišily pouze množinou pravidel.

5.1.2 Derivační módy

Derivační mód CD gramatického systému udává, jakým způsobem dochází k předávání řízení mezi jednotlivými gramatikami. Zavádí tedy pravidla, která určují, jak dlouho (kolik derivačních kroků) aktuálně aktivní gramatika může nebo musí provést, aby mohla předat řízení další gramatice, tj. komponentě systému.

Rozlišují se dva základní módy. První se nazývá ukončovací mód (anglicky *Terminating Mode*) a funguje velmi prostě. Každá komponenta pracuje (provádí přímé derivační kroky) tak dlouho, dokud může na větnou formu aplikovat některé ze svých pravidel. Tedy jednoduše řečeno „pracuje, dokud může" a poté předá řízení další komponentě v pořadí.

Naopak druhý mód je striktnější. Nazývá se k-krokový mód (anglicky *K-Step Mode*) a určuje nebo omezuje počet derivačních kroků, které musí každá komponenta dodržet. Zavádí tedy tři podvarianty, z nichž každá má svoje označení:

- **právě $\boldsymbol{k}$ kroků** $(= \boldsymbol{k})$ – každá komponenta musí provést přesný počet derivačních kroků, aby mohla předat řízení další komponentě. Pokud toto omezení nemůže splnit, pak daný jazyk není jazykem tohoto systému.
- **maximálně $\boldsymbol{k}$ kroků** $(\leq \boldsymbol{k})$ – tato varianta je nejmírnější, protože komponenta může provést libovolný počet deviračních kroků, který nepřesáhne stanovené omezení *k*.
- **alespoň $\boldsymbol{k}$ kroků** $(\geq \boldsymbol{k})$ – není tak striktní jako první varianta, ale vynucuje minimální počet derivačních kroků, které musí každá komponenta provést.

Okrajově je možné se setkat ještě se třetím módem, který je označován hvězdičkou $(*)$. Tento mód je nejbenevolentnější a říká, že každá komponenta může pracovat tak dlouho, dokud sama potřebuje, což je velmi nedeterministické.

Postupně byly vysvětleny základní principy jednotlivých derivačních módů CD gramatických systémů a nyní následuje ukázka formálních definic a způsob jejich zápisu.

Terminating Mode (ukončující mód)

$x_i \Rightarrow^t y$, x derivuje y v i-té komponentě v t-módu, právě když:

1. x derivuje y v G_i ($x \Rightarrow^* y$ v $G_i = (N, T, P_i, S)$) a
2. y nederivuje/neexistuje přímá derivace y pro jakékoliv z nad abecedou $N \cup T$, $y \not\Rightarrow z$ pro $\forall\ z \in (N \cup T)^*$

K-Step Mode

1. k-step derivace $x_i \Rightarrow^{=k} y$ pokud $x \Rightarrow^k y$ v G_i
2. nejvýše k-step derivace $x_i \Rightarrow^{\leq k} y$ pokud $x \Rightarrow^j y$ v G_i pro $j \leq k$
3. alespoň k-step derivace $x_i \Rightarrow^{\geq k} y$ pokud $x \Rightarrow^j y$ v G_i pro $j \geq k$

Obecně se provádí přímé derivační kroky, dokud nevznikne řetězec terminálů a nelze pokračovat dále, v tomto okamžiku výpočet končí.

5.1.3 Generovaný jazyk

Spojením všech derivačních módu dohromady vzniká množina derivačních módů D.

$$D = \{*, t\} \cup \{\leq k, = k, \geq k : k = 1, 2, \dots\}$$

Pokud se k množině derivačních módů přidá gramatika, vznikne množina všech možných derivací F.

$$F(G_j, u, f) = \{v : u_j \Rightarrow^f v\}, \text{ kde } j \in \{1, \dots, n\}, f \in D, u \in V^*$$

Pro lepší pochopení, V je množina všech řetězců takových, že se z řetězce u módem f v j-té komponentě provede derivace z u do v.

Nechť $\Gamma = (N, T, S, P_1, \dots, P_n)$ je CD gramatický systém v derivačním módu $f \in D$, potom jazyk generovaný tímto módem je následující:

$$L_f(\Gamma) = \{w \in T^* : \text{kde jsou } v_0, v_1, \dots, v_n \text{ tak, že}$$
$$v_i \in F(G_{j_i}, v_{-1}, f),\ i = 1, \dots, m,\ j_i \in \{1, \dots, n\},\ v_0 = S,\ v_m = w \text{ pro } m \geq 1\}.$$

5.1.4 Příklady

Nyní následuje několik základních příkladů pro praktickou ukázku, jak pracují CD gramatické systémy a jaké jazyky mohou generovat.

Příklad 1.: $L_t(\Gamma) = ?$ pro CD gramatický systém $\Gamma = (\{S, A\}, \{a\}, S, P_1, P_2, P_3)$ s komponentami:

$P_1 = \{S \rightarrow AA\}$

$$\boldsymbol{P_2 = \{A \to S\}}$$
$$\boldsymbol{P_3 = \{A \to a\}}$$

Ze zadání je na první pohled patrné, že se jedná o ukončující mód. Tedy každá komponenta bude pracovat tak dlouho, dokud může aplikovat některé ze svých pravidel. Výpočet musí zahájit první komponenta $\boldsymbol{P_1}$, protože žádná jiná nemá pravidla pro rozgenerování startujícího neterminálu. Provede tedy derivační krok z $\boldsymbol{S}$ do $\boldsymbol{AA}$ a musí skončit, nemá již žádné další pravidlo, které by mohla aplikovat.

$$\boldsymbol{S \Rightarrow_{P_1} AA}$$

Komponenta $\boldsymbol{P_1}$ bude mít možnost ovlivnit větnou formu pouze po činnosti komponenty $\boldsymbol{P_2}$, takže na bezprostředně dalším výpočtu se budou podílet komponenty $\boldsymbol{P_2}$ nebo $\boldsymbol{P_3}$. Komponenta $\boldsymbol{P_3}$ má sílu ukončit výpočet v okamžiku, kdy se celá větná forma skládá pouze z neterminálů $\boldsymbol{A}$, což je právě tento případ. Derivuje neterminály $\boldsymbol{A}$ na výsledné terminály, a protože v ukončujícím módu komponenta pracuje, dokud může, bude provádět derivace tak dlouho, dokud nezůstane řetězec terminálů, a výpočet tím skončí. Pro lepší ukázku bude tedy pracovat komponenta $\boldsymbol{P_2}$, po které bude muset převzít řízení opět komponenta $\boldsymbol{P_1}$.

$$\boldsymbol{AA \Rightarrow_{P_2} SA \Rightarrow_{P_2} SS \Rightarrow_{P_1} AAAA}$$

Pokud by nyní pokračovala komponenta $\boldsymbol{P_2}$, opakovala by se předchozí sekvence, proto bude ukázáno ukončení výpočtu komponentou $\boldsymbol{P_3}$.

$$\boldsymbol{AAAA \Rightarrow_{P_3} aAAA \Rightarrow_{P_3} aaAA \Rightarrow_{P_3} aaaA \Rightarrow_{P_3} aaaa}$$

Vznikl tedy řetězec složený ze čtyř terminálů $\boldsymbol{a}$, čímž výpočet končí. Je zřejmé, že podle toho, kolikrát bude pracovat komponenta $\boldsymbol{P_2}$, takovou délku bude mít výsledný řetězec terminálů $\boldsymbol{a}$, přičemž samotná délka je omezena na mocniny dvou. V nejjednodušším případě skončí výpočet po třech derivačních krocích, pokud po komponentě $\boldsymbol{P_1}$ bude pracovat rovnou komponenta $\boldsymbol{P_3}$ a výsledkem bude řetězec $\boldsymbol{aa}$. Výsledný jazyk tohoto systému je tedy:

$$\underline{\underline{\boldsymbol{L_t(\Gamma) = \{a^{2^n} : n \geq 1\}}}}$$

Příklad 2.: $\boldsymbol{L_{=2}(\Gamma)}$ = ? CD gramatický systém $\boldsymbol{\Gamma = (\{S, A, A', B, B'\}, \{a, b, c\}, S, P_1, P_2)}$, kde:

$$\boldsymbol{P_1 = \{S \to S,\ S \to AB,\ A' \to A,\ B' \to B\}}$$
$$\boldsymbol{P_2 = \{A \to aA'b,\ B \to cB',\ A \to ab,\ B \to c\}}$$

Tento systém je o něco složitější než v prvním příkladu. Obsahuje dvě komponenty, z nichž každá obsahuje čtyři pravidla. Zároveň se už nejedná o ukončující mód, ale jak je podle požadovaného jazyka vidět, systém se nachází v k-step módu. Konkrétně tedy v první variantě tohoto módu, a to 2-step módu $(= \boldsymbol{2})$. Každá komponenta musí provést přesně dva derivační kroky, aby mohla předat řízení druhé komponentě. Pokud by nemohla tuto podmínku splnit, pak končí neúspěchem.

Samotné derivace probíhají stejně, jak to bylo detailně ukázáno v prvním příkladu, proto zde nebude vysvětlen celý výpočet krok po kroku, ale bude uveden pouze základní princip a možný postup. Generování musí začít komponenta $\boldsymbol{P_1}$, která má opět jako jediná pravidla pro zpracování startujícího neterminálu. První pravidlo komponenty $\boldsymbol{P_1}$, $\boldsymbol{S \rightarrow S}$, je z praktického pohledu zbytečné, ale nesplnili bychom bez něho podmínku systému $(= \mathbf{2})$. Aplikují se tedy první a druhé pravidlo komponenty $\boldsymbol{P_1}$ a předá se řízení komponentě $\boldsymbol{P_2}$. Ta opět provede dva derivační kroky a vrátí řízení zpět komponentě $\boldsymbol{P_1}$. Vždy se musí dodržet omezující podmínky systému, a proto se aplikují buďto dvě terminální nebo dvě neterminální pravidla. Výsledkem bude řetězec složený popořadě ze stejného počtu písmen $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$. Délka řetězce se odvíjí od toho, kolikrát se komponenty prostřídají.

$$\boldsymbol{S \Rightarrow_{P_1} S \Rightarrow_{P_1} AB \Rightarrow_{P_2} aA'bB \Rightarrow_{P_2} aA'bcB' \Rightarrow_{P_1} aAbcB' \Rightarrow_{P_1} aAbcB \Rightarrow_{P_2}}$$
$$\boldsymbol{\Rightarrow_{P_2} aaA'bbcB \Rightarrow_{P_2} aaA'bbccB' \Rightarrow_{P_1} aaAbbccB' \Rightarrow_{P_1} aaAbbccB \Rightarrow_{P_2}} \dots$$
$$\boldsymbol{\Rightarrow_{P_1} a^nAb^nc^nB \Rightarrow_{P_2} a^{n+1}b^{n+1}c^nB \Rightarrow_{P_2} a^{n+1}b^{n+1}c^{n+1}}$$

$$\underline{\underline{\boldsymbol{L_{=2}(\Gamma) = \{a^nb^nc^n : n \geq 1\}}}}$$

Příklad 3.: $\boldsymbol{L_{=3}(\Gamma)}$ = ? pro CD gramatický systém z příkladu 2.

Závěrečný příklad ukazuje situaci, kdy není možné splnit zadané podmínky systému. Použije se stejný systém z předchozího příkladu a pouze se upraví podmínka z *přesně dvou kroků* na *tři kroky*. Vznikne tedy 3-step mód $(= \mathbf{3})$. Opět začne komponenta $\boldsymbol{P_1}$, ale po aplikaci prvních dvou pravidel není již v komponentě další pravidlo, které by bylo možné aplikovat, a nelze tak splnit podmínku přesně tří derivačních kroků pro každou komponentu. Stejný výsledek by byl i pro variantu $\boldsymbol{k} \geq \mathbf{3}$.

$$\boldsymbol{S \Rightarrow_{P_1} S \Rightarrow_{P_1} AB \Rightarrow_{P_1} \ neexistuje}$$

$$\underline{\underline{\boldsymbol{L_{=3}(\Gamma) = L_{\geq 3}(\Gamma) = \emptyset}}}$$

5.1.5 Třídy jazyků CD gramatických systémů a jejich generativní síla

Pro univerzální definování jednotlivých tříd jazyků CD gramatických systému bylo zavedeno následující značení:

$$\boldsymbol{CD_x^y(f)}, \text{ kde}$$

- $\boldsymbol{f}$ – derivační mód, $\boldsymbol{f \in D}$
- $\boldsymbol{y}$ – pokuch chybí, potom nepovoluje ε-pravidla
  – $\boldsymbol{\varepsilon}$, pak jsou ε-pravidla povolena
- $\boldsymbol{x}$ – $\boldsymbol{n}$, nejvyšší stupeň CD gramatického systému, $\boldsymbol{n \geq 1}$
  – ∞, počet komponent systému není omezen

$\boldsymbol{CD}_\infty(=)$ sjednocení všech tříd jazyků $\boldsymbol{CD}_\infty(=\boldsymbol{k})$ pro $\boldsymbol{k}=\boldsymbol{1},\boldsymbol{2},\ldots$
$\boldsymbol{CD}_\infty(\geq)$ sjednocení všech tříd jazyků $\boldsymbol{CD}_\infty(\geq\boldsymbol{k})$ pro $\boldsymbol{k}=\boldsymbol{1},\boldsymbol{2},\ldots$

Generativní síla CD gramatických systémů se liší podle zavedených omezení či počtu komponent. Následujících pět teorémů zachycuje vztahy jednotlivých variant CD gramatických systémů a jejich generativní síly vůči ostatním typům.

1. $\boldsymbol{CD}_\infty^{\boldsymbol{y}}(\boldsymbol{f})=\mathscr{L}(\boldsymbol{CF})$, pro všechny $\boldsymbol{f}\in\{=\boldsymbol{1},\geq\boldsymbol{1},*\}\cup\{\leq\boldsymbol{k}:\boldsymbol{k}\geq\boldsymbol{1}\}$
2. $\mathscr{L}(\boldsymbol{CF})=\boldsymbol{CD}_{\boldsymbol{1}}^{\boldsymbol{y}}(\boldsymbol{f})\subset\boldsymbol{CD}_{\boldsymbol{2}}^{\boldsymbol{y}}(\boldsymbol{f})\subseteq\boldsymbol{CD}_{\boldsymbol{r}}^{\boldsymbol{y}}(\boldsymbol{f})\subseteq\boldsymbol{CD}_\infty^{\boldsymbol{y}}(\boldsymbol{f})\subseteq\mathscr{L}(\boldsymbol{M})$,
   pro všechny $\boldsymbol{f}\in\{=\boldsymbol{k},\geq\boldsymbol{k}:\boldsymbol{k}\geq\boldsymbol{2}\}$, $\boldsymbol{r}\geq\boldsymbol{3}$
3. $\boldsymbol{CD}_{\boldsymbol{r}}^{\boldsymbol{y}}(\geq\boldsymbol{k})\subseteq\boldsymbol{CD}_{\boldsymbol{r}}^{\boldsymbol{y}}(\geq\boldsymbol{k}+\boldsymbol{1})$
4. $\boldsymbol{CD}_\infty^{\boldsymbol{y}}(\geq)\subseteq\boldsymbol{CD}_\infty^{\boldsymbol{y}}(=)$
5. $\mathscr{L}(\boldsymbol{CF})=\boldsymbol{CD}_{\boldsymbol{1}}^{\boldsymbol{y}}(\boldsymbol{t})=\boldsymbol{CD}_{\boldsymbol{2}}^{\boldsymbol{y}}(\boldsymbol{t})\subset\boldsymbol{CD}_{\boldsymbol{3}}^{\boldsymbol{y}}(\boldsymbol{t})=\boldsymbol{CD}_\infty^{\boldsymbol{y}}(\boldsymbol{t})=\mathscr{L}(\boldsymbol{ET0L})$

CD gramatický systém, který není omezen počtem komponent, připuštěním epsilon pravidel a který může použít libovolný k-step derivační mód s jedním či více kroky, má generativní sílu shodnou s třídou bezkontextových jazyků (teorém 1).

Zavede-li se omezení na k-step derivační mód se dvěma kroky a více (epsilon pravidla nejsou opět omezena), pak s počtem komponent roste generativní síla počínaje silou třídy bezkontextových jazyků a konče třídou maticových jazyků (teorém 2).

Pokud se vezmou dvě třídy jazyků CD gramatických systémů se stejným počtem komponent v módu „alespoň *k* kroků“, pak je třída s omezením na alespoň *k* kroků podmnožinou třídy s omezením alespoň (*k+1*) kroků (teorém 3).

Podobně je na tom třída jazyků CD gramatických systémů v módu „alespoň *k* kroků“, která je podmnožinou třídě v módu „přesně *k* kroků“ (teorém 4).

Pokud se jedná o ukončující mód, potom generativní síla tříd jazyků CD gramatických systémů v ukončujícím módu roste s počtem komponent. Při omezení na jednu či dvě komponenty dosahuje síly shodné s třídou bezkontextových jazyků. Od tří komponent a více je síla stejná s generativní sílou třídy ET0L jazyků (teorém 5). Podrobnější informace lze nalézt v publikaci [7].

5.1.6 Hybridní CD gramatické systémy

Zatímco klasické CD gramatické systémy definovaly jeden globální derivační mód pro všechny komponenty (gramatiky) systému, hybridní CD gramatické systémy umožňují definovat vlastní derivační mód pro každou komponentu zvlášť. Díky tomu lze každé komponentě na základě jejich pravidel a vhodnosti použití vybrat vlastní mód, ve kterém bude pracovat. Tedy jedna komponenta může pracovat v ukončujícím módu, zatímco druhá bude přepínat přesně po dvou krocích atd. Formální definice hybridního CD gramatického systému se liší pouze v rozšíření jednotlivých komponent o daný derivační mód.

$$\boldsymbol{\Gamma}=(\boldsymbol{N},\boldsymbol{T},\boldsymbol{S},(\boldsymbol{P_1},\boldsymbol{f_1}),\ldots,(\boldsymbol{P_n},\boldsymbol{f_n}))\text{, kde:}$$

- $\boldsymbol{N}$ – abeceda neterminálů
- $\boldsymbol{T}$ – abeceda terminálů, platí $\boldsymbol{N}\cap\boldsymbol{T}=\emptyset$

- $\boldsymbol{S}$ – startující symbol, $\boldsymbol{S} \in \boldsymbol{N}$
- $\boldsymbol{P_1} \dots \boldsymbol{P_n}$ – konečná množina bezkontextových pravidel tvaru $\boldsymbol{a} \rightarrow \boldsymbol{x},\ \boldsymbol{a} \in \boldsymbol{N},\ \boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T})^*$
- $\boldsymbol{f_1} \dots \boldsymbol{f_n}$ – mód i-té komponenty, $\boldsymbol{f_i} \in \boldsymbol{D}$ pro všechna $\boldsymbol{i} \in \{\boldsymbol{1}, \dots, \boldsymbol{n}\}$

Formální definice jazyka generovaného hybridním CD gramatickým systémem:

$$\boldsymbol{L_f}(\boldsymbol{\Gamma}) = \{\boldsymbol{w} \in \boldsymbol{T}^*\text{: kde jsou } \boldsymbol{v_0}, \boldsymbol{v_1}, \dots, \boldsymbol{v_n} \text{ tak, že}$$
$$\boldsymbol{v_i} \in \boldsymbol{F}(\boldsymbol{G_{j_i}}, \boldsymbol{v_{-1}}, \boldsymbol{f_j}),\ \boldsymbol{i} = \boldsymbol{1}, .., \boldsymbol{m},\ \boldsymbol{j_i} \in \{\boldsymbol{1}, \dots, \boldsymbol{n}\},\ \boldsymbol{v_0} = \boldsymbol{S},\ \boldsymbol{v_m} = \boldsymbol{w} \text{ pro nějaké } \boldsymbol{m} \geq \boldsymbol{1}\}.$$

5.1.7 Třídy jazyků hybridních CD gramatických systémů a jejich generativní síla

Obecný zápis pro jednotlivé třídy jazyků hybridních CD gramatických je rozšířením předchozí varianty pro klasické CD gramatické systémy:

$$\boldsymbol{XCD_{x,v}^{y}(f)}, \text{ kde}$$

- $\boldsymbol{f}$ – derivační mód, $\boldsymbol{f} \in \boldsymbol{D}$
- $\boldsymbol{y}$ – pokud chybí, potom nepovoluje ε-pravidla
  – $\boldsymbol{\varepsilon}$, pak jsou ε-pravidla povolena
- $\boldsymbol{x}$ – $\boldsymbol{n}$, nejvyšší stupeň CD gramatického systému, $\boldsymbol{n} \geq \boldsymbol{1}$
  – ∞, počet komponent systému není omezen
- $\boldsymbol{v}$ – $\boldsymbol{m}$, každá komponenta $\boldsymbol{P_i}$ obsahuje maximálně $\boldsymbol{m}$ pravidel, $\boldsymbol{m} > \boldsymbol{1}$
  – ∞ nebo nic, počet pravidel není omezen
- $\boldsymbol{X}$ – $\boldsymbol{nic}$, nedeterministický systém
  – $\boldsymbol{D}$, deterministický (pro každé pravidlo z $\boldsymbol{P_i}$ platí, že pro každý neterminál $\boldsymbol{A}$ existuje nejvýše jedna pravá strana; $\boldsymbol{A} \rightarrow \boldsymbol{u}$, $\boldsymbol{A} \rightarrow \boldsymbol{w} \in \boldsymbol{P_i}$, pak $\boldsymbol{u} = \boldsymbol{w}$)
  – $\boldsymbol{H}$, hybridní systém (nepíše se $(\boldsymbol{f})$, protože neexistuje globální mód)

I pro hybridní CD gramatické systémy lze uvést sadu teorémů, které demonstují jejich generativní sílů vůči ostatním trídám jazyků v rámci Chomského hierarchie jazyků. Rozbor jednotlivých teorémů je analogický s předchozím vysvětlením klasických CD gramatických systémů, pouze s rozdílem, že je nutné sledovat jednotlivé atributy z definice tříd hybridních CD gramatických systémů.

1. $\boldsymbol{CD_{\infty,\infty}(f)} = \boldsymbol{CD_{\infty,1}(f)} = \mathscr{L}(\boldsymbol{CF})$, pro všechny $\boldsymbol{f} \in \{=\boldsymbol{1}, \geq \boldsymbol{1}, *\} \cup \{\leq \boldsymbol{k}: \boldsymbol{k} \geq \boldsymbol{1}\}$
2. $\mathscr{L}(\boldsymbol{CF}) \subset \boldsymbol{CD_{\infty,1}^{\varepsilon}(t)} \subset \boldsymbol{CD_{\infty,2}^{\varepsilon}(t)} \subseteq \boldsymbol{CD_{\infty,3}^{\varepsilon}(t)} \subseteq \boldsymbol{CD_{\infty,4}^{\varepsilon}(t)} \subseteq \boldsymbol{CD_{\infty,5}^{\varepsilon}(t)} = \boldsymbol{CD_{\infty,\infty}^{\varepsilon}(t)} =$ $= \mathscr{L}(\boldsymbol{ETOL})$
3. $\boldsymbol{CD_{n,m}(f)} \subset \boldsymbol{CD_{n+1,m}(f)},\ \boldsymbol{f} \in \{*, \boldsymbol{t}\}$
4. $\boldsymbol{CD_{n,m}(f)} \subset \boldsymbol{CD_{n,m+1}(f)},\ \boldsymbol{f} \in \{*, \boldsymbol{t}\}$
5. $\mathscr{L}(\boldsymbol{CF}) = \boldsymbol{HCD_1} \subset \boldsymbol{HCD_2} \subseteq \boldsymbol{HCD_3} \subseteq \boldsymbol{HCD_4} = \boldsymbol{HCD_\infty} = \mathscr{L}(\boldsymbol{M})$
6. $\mathscr{L}(\boldsymbol{ETOL}) \subset \boldsymbol{HCD_4}$
7. $\boldsymbol{CD_\infty}(=) \subset \boldsymbol{HCD_3}$

5.2 PC gramatické systémy

Paralelně komunikující gramatické systémy, zkráceně PC gramatické systémy (z angličtiny *Parallel Communicating Grammar Systems*), jak již z názvu vyplývá, pracují paralelně. To je zásadní rozdíl oproti CD gramatickým systémům, které pracují sekvenčně a postupně se střídají ve zpracování jedné společné větné formy. U PC gramatických systémů pracují všechny komponenty současně na své vlastní větné formě. Důležité je i druhé slovo názvu, „komunikující", protože nejenže komponenty pracují paralelně, ale mohou mezi sebou i komunikovat pomocí tzv. komunikačních symbolů. Tímto způsobem si mezi sebou předávají mezivýsledky nebo si může jedna komponenta vyžádat práci jiné komponenty. Tohoto principu se využívá např. při syntaktické analýze. V mnoha případech může být užitečné a velmi praktické mít pro analýzu či překlad více různých gramatik, ať už pro různé části vstupního kódu nebo obecně může být analyzovaný kód tvořen více různými jazyky. Pak je vhodné aplikovat v průběhu analýzy odpovídající gramatiky. Zároveň však při komunikaci mezi komponentami hrozí riziko uváznutí, tzv. deadlock, kterému je nutné se vyvarovat.

5.2.1 Formální definice

PC gramatický systém $\boldsymbol{\Gamma}$ stupně $\boldsymbol{n}$, $\boldsymbol{n \geq 1}$, je $(n+3)$-tice stejně jako CD gramatický systém, avšak svým zápisem je spíše podobný hybridnímu CD gramatickému systému. $\boldsymbol{n}$ značí opět počet množin pravidel systému, přičemž každá množina $\boldsymbol{P}$ má svůj vlastní startovací neterminál.

$$\boldsymbol{\Gamma} = (\boldsymbol{N}, \boldsymbol{K}, \boldsymbol{T}, (\boldsymbol{S_1}, \boldsymbol{P_1}), \dots, (\boldsymbol{S_n}, \boldsymbol{P_n})), \text{ kde:}$$

- $\boldsymbol{N}$ – abeceda neterminálů
- $\boldsymbol{K}$ – konečná množina komunikačních symbolů, $\boldsymbol{K} \in \{\boldsymbol{Q_1}, \dots, \boldsymbol{Q_n}\}$
- $\boldsymbol{T}$ – abeceda terminálů
- $\boldsymbol{S_i}$ – startující symbol i-té komponenty, $\boldsymbol{S_i} \in \boldsymbol{N}$ **pro** $\boldsymbol{i = 1, .., n}$
- $\boldsymbol{P_i}$ – konečná množina pravidel tvaru $\boldsymbol{A \to x}$**,** kde $\boldsymbol{A} \in \boldsymbol{N}$ **a** $\boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T} \cup \boldsymbol{K})^*$ pro $\boldsymbol{i = 1, \dots, n}$
- Dále platí, že $\boldsymbol{K} \cap \boldsymbol{N} = \emptyset$, $\boldsymbol{K} \cap \boldsymbol{T} = \emptyset$

5.2.2 Derivační kroky

U PC gramatických systémů se nerozlišují derivační módy, jako tomu bylo u CD gramatických systémů, ale při výpočtu se střídají dva druhy derivačních kroků. Oba druhy derivačních kroků jsou velmi specifické a mají svůj speciální účel. Primárním derivačním krokem, dalo by se říci i režimem, je tzv. ***generující krok*** (g-Step), během kterého systém provádí klasické generativní kroky z $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})$ do $(\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$, tedy $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})$ přímo derivuje $(\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$. Tento režim je totožný s derivačními kroky, které provádí CD gramatické systémy. N-tice $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})$ a $(\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$, kde $\boldsymbol{x_i}, \boldsymbol{y_i} \in (\boldsymbol{N} \cup \boldsymbol{T} \cup \boldsymbol{K})^*$ pro všechna $\boldsymbol{i}$: $\boldsymbol{1 \leq i \leq n}$, se nazývají konfigurace systému.

Zatím se PC gramatické systémy, až na paralelní provádění, tolik neliší od CD gramatických systémů. Významný rozdíl nastává při přepnutí systému do druhého režimu, tzv. ***komunikační krok*** (c-Step). K tomu dochází v okamžiku, kdy některá z komponent vygeneruje komunikační symbol. Zastaví se provádění generujících kroků ve všech komponentách a spustí se obluha komunikačního

kroku, ve které se provede přepsání vygenerovaného komunikačního symbolu uvnitř větné formy (konfigurace) komponenty, která tento symbol vygenerovala, řetězcem (konfigurací) komponenty, na kterou komunikační symbol odkazuje. Odkazovaná komponenta však nesmí mít ve své větné formě žádné komunikační symboly. Určitě se to zdá trochu zmatené, ale dále následuje definice jednotlivých režimů a příklady, ze kterých bude jejich fungování pochopitelnější. Důležité je, že komunikační krok (režim) má vyšší prioritu než generativní krok, takže jsou vždy nejdříve zpracovány komunikační symboly a až poté systém pokračuje v generativním kroku. Lze si tedy komunikační krok představit jako obsluhu přerušení, pokud se uvažuje analogii z operačních systémů.

g-Step

Pokud $\boldsymbol{x_i} \Rightarrow \boldsymbol{y_i}$ v $\boldsymbol{G_i} = (\boldsymbol{N} \cup \boldsymbol{K}, \boldsymbol{T}, \boldsymbol{P_i}, \boldsymbol{S_i})$ nebo

$\boldsymbol{x_i} = \boldsymbol{y_i} \in \boldsymbol{T}^*$ pro všechny $\boldsymbol{1} \leq \boldsymbol{i} \leq \boldsymbol{n}$, pak

$$(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})_{\boldsymbol{g}} \Rightarrow (\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$$

c-Step

Nastav $\boldsymbol{z_i} = \boldsymbol{x_i}$ pro všechny $\boldsymbol{i} = \boldsymbol{1}, \dots, \boldsymbol{n}$

Pro každé $\boldsymbol{i} = \boldsymbol{1}, \dots, \boldsymbol{n}$ pokud

$\mathbf{abeceda}(\boldsymbol{x_i}) \cap \boldsymbol{K} \neq \emptyset$

a zároveň pro každé $\boldsymbol{Q_j}$ v $\boldsymbol{x_i}$

$\mathbf{abeceda}(\boldsymbol{x_j}) \cap \boldsymbol{K} = \emptyset$

potom pro každé $\boldsymbol{Q_j}$ v $\boldsymbol{x_i}$

1. Nastav $\boldsymbol{z_j} = \boldsymbol{S_j}$
2. Nahraď $\boldsymbol{Q_j}$ za $\boldsymbol{x_j}$ v $\boldsymbol{x_i}$
3. Nastav do $\boldsymbol{z_i}$ výsledek z kroku 2

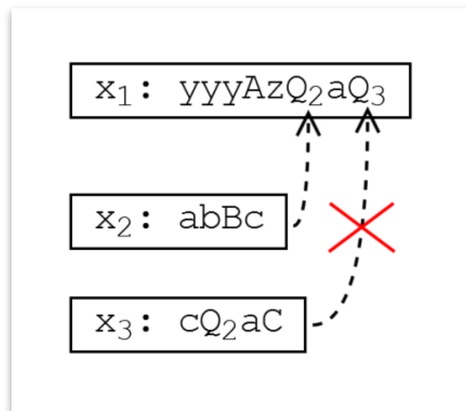
Proveď

$(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})_{\boldsymbol{c}} \Rightarrow (\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$

pomocí $\boldsymbol{y_i} = \boldsymbol{z_i}$ pro všechny $\boldsymbol{i} = \boldsymbol{1}, \dots, \boldsymbol{n}$

Při generativním kroku, jak již bylo zmíněno výše, se provádí klasické derivační kroky z $\boldsymbol{x_i}$ do $\boldsymbol{y_i}$ pro všechny komponenty. V případě, že $\boldsymbol{x_i}$ již tvoří řetězce terminálů, provede se pouze simulace derivačního kroku, ale větná forma dané komponenty zůstane nezměněna. Ostatní komponenty mohou provést různé změny svých větných forem, zatímco tato „čeká“.

Naproti tomu komunikační krok provádí naprosto odlišné úkony a jeho algoritmus je daleko složitější, proto zde bude důkladně vysvětlen. Při úvodní inicializaci se do pole pomocných proměnných ($\boldsymbol{z_i}$) uloží větné formy ($\boldsymbol{x_i}$, pro všechna $\boldsymbol{i} = \boldsymbol{1}, \dots, \boldsymbol{n}$) jednotlivých komponent. Následně se v hlavním těle algoritmu procházejí větné formy. Pokud některá větná forma ($\boldsymbol{x_i}$) obsahuje jeden či více komunikačních symbolů ($\boldsymbol{Q_j}$) a zároveň komponenta, na kterou tento symbol odkazuje ($\boldsymbol{x_j}$), neobsahuje žádný komunikační symbol, pak se pro tuto komponentu může vykonat komunikační krok. První podmínka je jasná, nelze provádět komunikační krok v komponentě, která neobsahuje žádný komunikační symbol. Druhá podmínka předchází zanořování při zpracování a především možnému zacyklení, protože pokud by komponenta ($\boldsymbol{x_j}$), na kterou odkazuje komunikační symbol ($\boldsymbol{Q_j}$) obsahovala také komunikační symboly, musely by se pro ni také v nejbližších iteracích vykonat komunikační kroky a v případě, že by komponenta odkazovala sama na sebe, došlo by k zacyklení.



Obrázek 5: Příklad porušení podmínky výskytu K symbolů. Požadavek Q3 nebude v tomto kroku uspokojen.

Pokud je tedy podmínka splněna, potom se pro každou komponentu ($\boldsymbol{x_j}$) nejdříve upraví pomocné pole tak, že do odpovídajících proměnných nastaví startující symbol příslušné komponenty ($\boldsymbol{z_j} = \boldsymbol{S_j}$). Tento krok se provádí u PC gramatických systému s návraty (jeho další varianty budou vysvětleny později). Dále je to již přímočaré, přepíší se všechny komunikační symboly ($\boldsymbol{Q_j}$) v původní větné formě ($\boldsymbol{x_i}$) odpovídajícími větnými formami ($\boldsymbol{x_j}$) a výsledek se uloží do ($\boldsymbol{z_i}$). Až se zpracují všechny komponenty a komunikační symboly v tomto komunikačním kroku, přepíše se celé pomocné pole ($\forall\, \boldsymbol{z_i}$) na výstup (do proměnných $\boldsymbol{y_i}$).

5.2.3 Generovaný jazyk

PC gramatický systém, jak již bylo uvedeno výše, provádí buďto generativní kroky, které odpovídají klasickým derivačním krokům, nebo speciální komunikační kroky, které se snaží vypořádat s komunikačními symboly. Formálně lze generativní krok zapsat jako přechod z konfigurace $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})$ do konfigurace $(\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$, tedy $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})_{\boldsymbol{g}} \Rightarrow (\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$, u komunikačního kroku je zápis obdobný, $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})_{\boldsymbol{c}} \Rightarrow (\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$. Obecně tedy konfigurace $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n})$ provádí přímý derivační krok do konfigurace $(\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$, $(\boldsymbol{x_1}, \dots, \boldsymbol{x_n}) \Rightarrow (\boldsymbol{y_1}, \dots, \boldsymbol{y_n})$. Vychází-li se z počáteční konfigurace $(\boldsymbol{S_1}, \dots, \boldsymbol{S_n})$, která derivuje konfiguraci $(\boldsymbol{x}, \boldsymbol{\alpha_2}, \dots, \boldsymbol{\alpha_n})$, pak je výsledným jazykem generovaným tímto PC gramatickým systémem množina všech řetězců terminálů, které se vygenerují v první komponentě $\boldsymbol{x}$, ostatní komponenty $(\boldsymbol{\alpha_2}, \dots, \boldsymbol{\alpha_n})$ mohou obsahovat libovolné řetězce.

$$\boldsymbol{L}(\boldsymbol{\Gamma}) = \{\boldsymbol{x} \in \boldsymbol{T}^*: (\boldsymbol{S_1}, \dots, \boldsymbol{S_n}) \Rightarrow^* (\boldsymbol{x}, \boldsymbol{\alpha_2}, \dots, \boldsymbol{\alpha_n}),\ \boldsymbol{\alpha_i} \in (\boldsymbol{N} \cup \boldsymbol{T} \cup \boldsymbol{K})^*, \text{pro všechny } \boldsymbol{i} = \boldsymbol{2}, \dots, \boldsymbol{n}\}$$

5.2.4 Varianty PC gramatických systémů

Centralizovaný PC gramatický systém je jedním ze speciálních případů klasického PC gramatického systému. Zavádí omezení, že pouze první komponenta $\boldsymbol{P_1}$ může generovat komunikační symboly. Toto omezení má za přínos prevenci uváznutí, protože pouze do první komponenty mohou být vkládány větné formy ostatních komponent místo příslušných komunikačních symbolů a nemůže se stát, že by požadavek nemohl být uspokojen, protože by jiná komponenta obsahovala komunikační symboly. Celý systém je tedy řízen první komponentou, což se formálně zapíše následovně:

Nechť $\boldsymbol{\Gamma} = (\boldsymbol{N}, \boldsymbol{K}, \boldsymbol{T}, (\boldsymbol{S_1}, \boldsymbol{P_1}), \dots, (\boldsymbol{S_n}, \boldsymbol{P_n}))$ je PC gramatický systém. $\boldsymbol{\Gamma}$ je centalizovaný, pokud pro všechny $\boldsymbol{A} \to \boldsymbol{x} \in \boldsymbol{P_i}$, kde $\boldsymbol{i} = \boldsymbol{2}, \dots, \boldsymbol{n}$, **abeceda(x)** $\cap \boldsymbol{K} = \emptyset$.

Ve výkladu algoritmu komunikačního kroku byl zmíněn systému s návraty. Dle definice algoritmu se jedná o výchozí režim a vzniká v bodě 1 $(\boldsymbol{z_j} = \boldsymbol{S_j})$, kdy dojde k přepsání kopírované komponenty jejím startujícím neterminálem. U systému s návraty tedy nedochází pouze k nahrazení komunikačního symbolu v aktuálně zpracovávané komponentě, ale je zároveň i „resetována" komponenta, která je kopírována. Může se pak v praxi stát, že první komponenta již obsahuje řetězec terminálů, který patří do jazyka tohoto systému, ale jiná komponenta si vyžádá jeho vkopírování (vygeneruje odpovídající komunikační symbol – toto je možné pouze u necentralizovaného systému), tím je první komponenta nahrazena startujícím neterminálem a její rozgenerování začíná od začátku. Jak je patrné z tohoto příkladu, lze kombinovat ne/centralizovaný systém s režimem s návraty a bez návratů. Jazyk generovaný systémem s režimem s návraty se značí $\boldsymbol{L_r}(\boldsymbol{\Gamma})$.

Naopak PC gramatický systém bez návratů „neresetuje" kopírovanou komponentu startujícím symbolem, ale pokračuje ve zpracování stávající větné formy. Je však nutné upravit definici algoritmu pro komunikační krok a odstranit z ní bod 1 $(\boldsymbol{z_j} = \boldsymbol{S_j})$. Jazyk generovaný tímto systémem se značí $\boldsymbol{L_{nr}}(\boldsymbol{\Gamma})$.

5.2.5 Příklad

Příklad 1.: $\boldsymbol{L_r}(\boldsymbol{\Gamma})$ =? pro PC gramatický systém

$$\boldsymbol{\Gamma} = (\{\boldsymbol{S_1}, \boldsymbol{S'_1}, \boldsymbol{S_2}, \boldsymbol{S_3}\},\ \boldsymbol{K},\ \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\},\ (\boldsymbol{S_1}, \boldsymbol{P_1}),\ (\boldsymbol{S_2}, \boldsymbol{P_2}),\ (\boldsymbol{S_3}, \boldsymbol{P_3})\)$$ s komponentami:

$$\boldsymbol{P_1} = \{\boldsymbol{S_1} \to \boldsymbol{abc},\ \boldsymbol{S_1} \to \boldsymbol{a^2b^2c^2},\ \boldsymbol{S_1} \to \boldsymbol{aS'_1},\ \boldsymbol{S_1} \to \boldsymbol{a^3Q_2},\ \boldsymbol{S'_1} \to \boldsymbol{aS'_1},\ \boldsymbol{S'_1} \to \boldsymbol{a^3Q_2},\ \boldsymbol{S_2} \to \boldsymbol{b^2Q_3},\ \boldsymbol{S_3} \to \boldsymbol{c}\}$$

$$\boldsymbol{P_2} = \{\ \boldsymbol{S_2} \to \boldsymbol{b}\,\boldsymbol{S_2}\}$$

$$\boldsymbol{P_3} = \{\ \boldsymbol{S_3} \to \boldsymbol{c}\,\boldsymbol{S_3}\}$$

Ze zadání lze na první pohled zjistit některé vlastnosti systému. Index $\boldsymbol{r}$ určuje, že se jedná o systém s návraty. Po bližším prozkoumání jednotlivých pravidel je patrné, že komunikační symboly generuje pouze první komponenta, takže se tedy jedná o centralizovaný systém. Samotný výpočet začíná obdobně jako u CD gramatických systémů, pouze s tím rozdílem, že je nutné provést derivaci paralelně pro každou komponentu. U komponenty $\boldsymbol{P_1}$ dochází při startu k výběru ze tří pravidel, ale aby nebyl výpočet ukončen hned při prvním kroku, použije se pro ukázku třetí pravidlo. Naopak u komponent $\boldsymbol{P_2}$ a $\boldsymbol{P_3}$ je generování přímočaré, obě komponenty mají každá pouze po jednom pravidle, které neustále produkují nové terminály. Start bude tedy následující:

$$(\boldsymbol{S_1}, \boldsymbol{S_2}, \boldsymbol{S_3}) \Rightarrow (\boldsymbol{aS'_1}, \boldsymbol{b}\,\boldsymbol{S_2}, \boldsymbol{c}\,\boldsymbol{S_3})$$

Nevygeneroval se žádný komunikační symbol, takže systém zůstává v generativním kroku. Každá z komponent má „svůj" terminál, který může aplikací vhodných pravidel generovat do nekonečna. Provede se tedy simulace nekonečného generování, aby byly pokryty všechny možnosti výsledného jazyka. Pro komponentu $\boldsymbol{P_1}$ to znamená opakovat páté pravidlo, pro komponenty $\boldsymbol{P_2}$ a $\boldsymbol{P_3}$ stačí opakovat jejich jediná pravidla. Výsledkem jsou tak n-té mocniny jednotlivých terminálů:

$$(aS_1', b\,S_2, c\,S_3) \Rightarrow^* (a^n S_1', b^n\,S_2, c^n\,S_3)$$

Nyní je nutné pro komponentu P_1 vybrat jiné pravidlo, aby se výpočet posunul dále. Jediná možnost je šesté pravidlo v pořadí. Zbylé komponenty pokračují stejným způsobem.

$$(a^n S_1', b^n\,S_2, c^n\,S_3) \Rightarrow (a^{n+3} Q_2, b^{n+1}\,S_2, c^{n+1}\,S_3)$$

Je vidět, že se v komponentě P_1 vygeneroval první komunikační symbol. Systém musí tedy přejít do komunikačního kroku a pokusit se tento komunikační symbol zpracovat. Jedná se o symbol Q_2, což znamená, že odkazuje na komponentu P_2, a vzhledem k tomu, že komponenta P_2 neobsahuje žádný komunikační symbol, nic nebrání v provedení komunikačního kroku. Důležité je ještě připomenout, že se jedná o systém s návraty, takže symbol Q_2 bude nahrazen větnou formou komponenty P_2, a ta bude následně „resetována“ startujícím neterminálem.

$$(a^{n+3} Q_2, b^{n+1}\,S_2, c^{n+1}\,S_3) \Rightarrow (a^{n+3} b^{n+1}\,S_2,\ S_2, c^{n+1}\,S_3)$$

Tímto byly vyřešeny všechny komunikační symboly, tedy jeden, a následuje generativní krok. V komponentě P_1 je možné aplikovat pouze sedmé pravidlo, které opět vygeneruje komunikační symbol. V tomto případě se však jedná o požadavek na komponentu P_3, protože symbol je Q_3. Opět dojde k přechodu do komunikačního kroku, symbol Q_3 bude nahrazen větnou formou komponenty P_3, a ta následně „resetována“ svým startujícím neterminálem.

$$(a^{n+3} b^{n+1}\,S_2,\ S_2, c^{n+1}\,S_3) \Rightarrow (a^{n+3} b^{n+3} Q_3,\ bS_2, c^{n+2}\,S_3) \Rightarrow (a^{n+3} b^{n+3} c^{n+2}\,S_3,\ bS_2,\ S_3)$$

Nyní lze v komponentě P_1 aplikovat pouze poslední pravidlo, čímž generování končí, protože vznikl řetězec terminálů. Zároveň se systém nachází v centralizovaném režimu, takže žádná jiná komponenta nemůže „resetovat“ první komponentu.

$$(a^{n+3} b^{n+3} c^{n+2}\,S_3,\ bS_2,\ S_3) \Rightarrow (a^{n+3} b^{n+3} c^{n+3},\ b^2 S_2, c\,S_3)$$

Na závěr musí být jazyk doplněn o výsledky pravidel, kterými mohl systém ukončit generování hned v prvních krocích, a výsledný jazyk bude vypadat následovně:

$$\underline{\underline{L_r(\Gamma) = \{a^n b^n c^n : n \geq 1\}}}$$

5.2.6 Třídy jazyků PC gramatických systémů a jejich generativní síla

Pro PC gramatické systémy bylo zavedeno následující univerzální značení jednotlivých typů systémů a jejich režimů:

XPC_nY, kde

- $\boldsymbol{X}$ – $\boldsymbol{N}$, systém bez návratů (non-returning mode)
  – $\boldsymbol{C}$, centralizovaný PC gramatický systém
- $\boldsymbol{n}$ – počet komponent systému, $\boldsymbol{n \geq 1}$
- $\boldsymbol{Y}$ – typ pravidel (REG, LIN, CF)

Vzorové příklady: $\boldsymbol{CPC_2REG}$, $\boldsymbol{NPC_2LIN}$, $\boldsymbol{NCPC_\infty}$

Následující teorémy zachycují několik vztahů mezi třídami jazyků PC gramatických systému a ostatních typů tříd jazyků:

1. $\boldsymbol{PC_nREG \subset PC_{n+1}REG}$ pro $\boldsymbol{n > 1}$
2. $\boldsymbol{CPC_nREG \subset CPC_nLIN \subset CPC_nCF}$ pro $\boldsymbol{n > 1}$
3. $\boldsymbol{NPC_\infty CF \subseteq PC_\infty CF}$
4. $\mathscr{L}(\mathbf{M}) \subset \boldsymbol{PC_\infty CF}$
5. $\mathscr{L}(\mathbf{LIN}) \subset \boldsymbol{PC_\infty REG}$

První teorém popisuje vzrůstající sílu PC gramatických systémů s regulárními pravidly při zvětšujícím se počtu komponent. Čím více komponent je použito, tím větší se získá generativní síla. Druhý teorém není nijak překvapivý, popisuje vztah mezi centralizovanými PC gramatickými systémy s $\boldsymbol{n}$ komponentami, přičemž čím silnější třída pravidel se použije, tím silnější třída jazyků je získána. Třetí teorém uvažuje neomezený počet komponent a říká, že při použití bezkontextových pravidel je třída jazyků u systému bez návratů podmnožinou třídy jazyků systému s návraty. Zbylé dva teorémy porovnávají maticové a lineární jazyky s PC gramatickými sytémy s neomezeným počtem komponent a bezkontextovými, resp. regulárními pravidly. Podrobnější informace s dalšími teorémy a důkazy lze nalézt v publikaci [7].

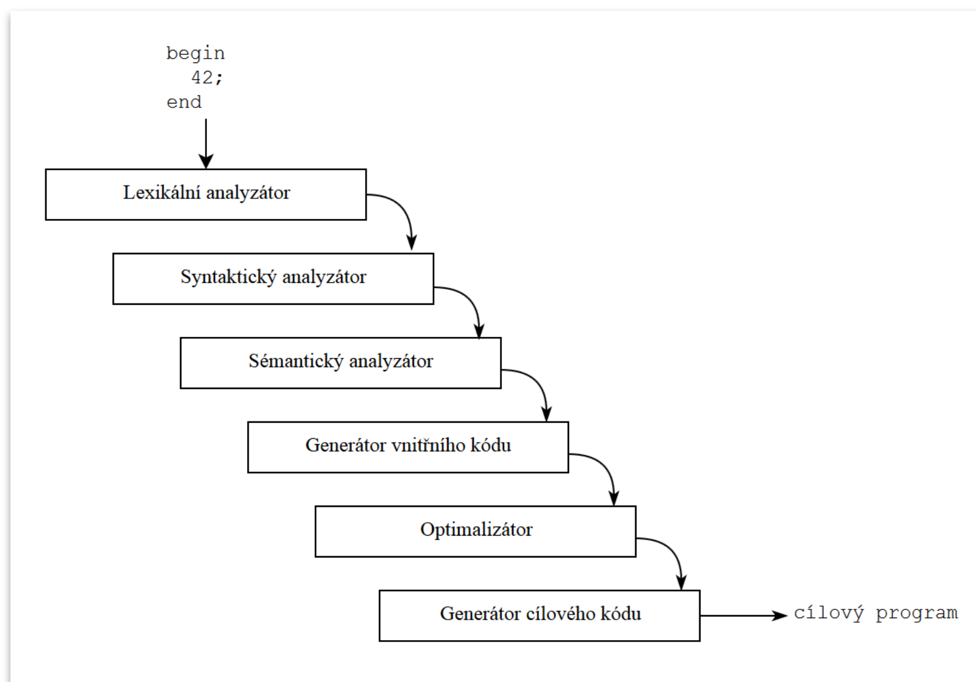
6 Syntaxí řízený překlad

Předchozí kapitoly pokrývaly především teorii. Práce je určená pro odborné čtenáře, ale přesto se úvod věnuje shrnutí obecnějšího základu, který by sice měl čtenář dobře znát, ale pro rychlé pochopení složitější látky je vhodné sjednotit i ty nejzákladnější stavební kameny. Nejdříve byly zopakovány obecné formální prostředky. Po nich následovaly regulární gramatiky, kterých se tato práce dotkne pouze v rámci implementace. Tvoří jádro lexikální analýzy, která je nezbytná jako přípravná fáze syntaktické analýzy. Poté následovala problematika bezkontextových gramatik a zásobníkových automatů. Bezkontextové gramatiky jsou velmi významné nejen z teoretického pohledu, ale umoňují definici a popis programovacích jazyků, takže jejich uplatnění v oblasti překladačů je velké. Následně již mohla navázat stěžejní část práce, a to samotné gramatické systémy, které formálně popsují součinnost gramatik a zavádějí různé modely jejich spolupráce.

Nyní je možné bezpečně přejít od teorie k praxi a zaměřit se na syntaxí řízený překlad a především pak jeho podčást, syntaktickou analýzu. Samotná syntaktická analýza slouží jako nástroj kontroly správné (syntaktické) struktury analyzovaného textu. Provádí se za pomoci požadované gramatiky definující jazyk, do kterého by měl kontrolovaný text patřit. Uvažujme tedy např. abecedu $\boldsymbol{\Sigma} = \{\boldsymbol{begin},\ ;,\ \boldsymbol{int},\ \boldsymbol{id},\ \boldsymbol{end}, \dots\}$, gramatiku $\boldsymbol{G}$, které odpovídá výsledný programovací jazyk $\boldsymbol{L_G}$, a

zkoumáme, zda-li řetězec $\boldsymbol{x}$ = „`begin 42; end`“ patří do daného jazyka, $\boldsymbol{x} \in \boldsymbol{L_G}$, a tedy se jedná o syntakticky správně napsaný zdrojový kód, či nikoli, $\boldsymbol{x} \notin \boldsymbol{L_G}$. Syntaktická analýza tvoří základ překladu jazyků, a pokud je v celém procesu zapojena jako řídící člen, jedná se o syntaxí řízený překlad. Překladač je obecně nástroj, který převádí zdrojový text (program napsaný ve zdrojovém jazyce) do podoby cílového textu (programu napsaného v cílovém jazyce), aniž by se změnil sémantický význam textu. U programů to znamená, že jsou oba programy, jak zdrojový, tak cílový, funkčně ekvivalentní. Zpravidla je v praxi žádoucí převést program z programovacího jazyka s vysokou abstrakcí, který je pro člověka dobře čitelný, do jazyka symbolických adres nebo přímo strojového kódu, aby ho mohl počítač snadno vykonávat.

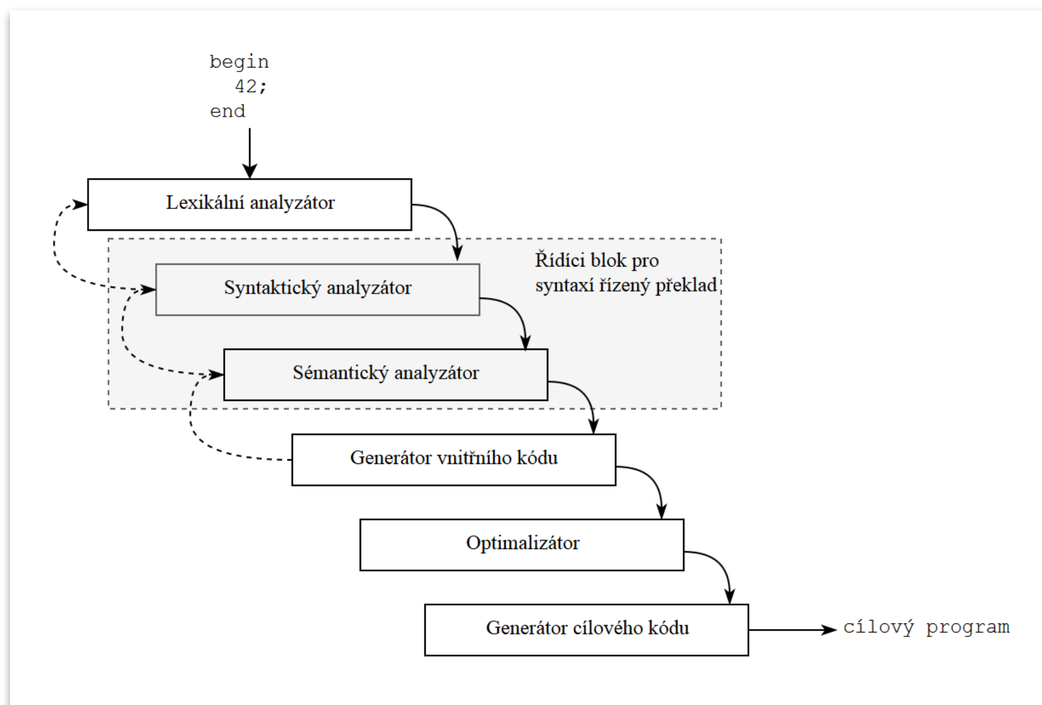
Obecný překladač si lze představit jako posloupnosti šesti na sebe navazujících bloků, které postupně zpracovávají vstupní text. Výstup prvního bloku je vstupem druhého bloku atd., až konečně poslední blok vygeneruje požadovaný cílový program.



Obrázek 6: Schéma obecného překladače

S ohledem na navazující praktickou část se tato práce zaměřuje pouze na syntaktickou analýzu, která je podmnožinou celého překladače a sestává se z prvních dvou bloků, tj. lexikálního analyzátoru a syntaktického analyzátoru. Další části překladače provádějí podrobnější kontroly kódu a připravují vše potřebné pro vygenerování cílového programu. Pokud je syntaxe vstupu v pořádku, potom sémantický analyzátor kontroluje datové typy, deklarace a inicializace proměnných a další náležitosti, na které již samotná syntaktická analýza nestačí. Následně zpravidla probíhá překlad do vnitřního (např. tříadresného) kódu, který je většinou univerzálně přenositelný a snadněji se nad ním provádějí volitelné optimalizace a generování výsledného kódu.

Pokud se jedná o syntaxí řízený překlad, bývají jednotlivé části překladače úzce propojené a komunikují spolu obousměrně během celého překladu, přičemž řídícím členem je syntaktický analyzátor.



Obrázek 7: Schéma překladače se syntaxí řízeným překladem

Při syntaxí řízeném překladu provádí syntaktický analyzátor kontrolu syntaktické struktury kódu a přitom volá a řídí ostatní bloky tak, aby byla aktuálně zpracovávaná část kódu rovnou zkontrolována i sémantickým analyzátorem a případně byl hned vygenerován odpovídající vnitřní kód. Tento přístup řízení je v praxi velmi častý. Není nutné, aby na sebe jednotlivé části překladače čekaly, ale analýza probíhá na více úrovních a umožňuje nejen úsporu času, ale i dřívější odhalení sémantických chyb, které by se jinak zjistily až po provedení celé syntaktické analýzy.

Lexikální analýza je nezbytná pro předpřipravení vstupního textu. Jejím vstupem je zdrojový program a výstupem řetězec tzv. tokenů. Jedná se o lexémy, tedy logické rozdělení vstupního kódu na lexikální jednotky jako jsou identifikátory, čísla, klíčová slova, operátory atd. Lexikální analýza vychází z povolené struktury vstupního textu (např. z jakých znaků a symbolů se může skládat identifikátor) a datových typů. Výsledné tokeny potom obsahují kromě samotného lexému i dodatečné informace, tzv. atributy, které usnadňují zbylý překlad. Lexikální analýza předchozího příkladu by mohla vypadat následovně:

Zdrojový program	**Identifikace lexémů**	**Výsledné tokeny**
`Begin`	`begin`	`begin` [keyword]
`    42;`	`42`	`42` [int]
`End`	`;`	`;`
	`end`	`end` [keyword]

Samotná implementace se provádí konečnými automaty, které byly právě z tohoto důvodu uvedeny na začátku práce.

Syntaktický analyzátor pracuje nad tokeny, které mu předá lexikální analyzátor, a snaží se k nim dle pravidel gramatiky sestavit odpovídající derivační strom. Pokud je takový strom nalezen, potom je vstupní text syntakticky správně. V opačném případě se jedná a syntaktickou chybu a

překlad končí. Pro sestavení derivačního stromu se uplatňují dva přístupy, a to shora dolů a zdola nahoru, které budou podrobně vysvětleny v následujících kapitolách.

Gramatické systémy lze zapojit do syntaktické analýzy mnoha způsoby dle jejich jednotlivých typů. Tato práce přistoupuje k dané problematice z obecného hlediska a pokouší se aplikovat znalosti gramatických systémů na obecně známou kombinaci přístupů syntaktické analýzy shora dolů s analýzou zdola nahoru. Každý z těchto přístupů má svá specifika a je vhodnější pro jiný způsob použití. Pro účel této práce byla aplikována analýzu shora dolů pro syntaktickou analýzu strukrury kódu, zatímco analýzu zdola nahoru pro zpracování matematických výrazů. Oba přístupy vyžadují vstupní text, neboli zdrojový kód, rozdělený na jednotlivé tokeny a hlavní rozdíl spočívá v tom, že se při syntaktické analýze shora dolů používá levý rozbor, tj. posloupnost pravidel nejlevější derivace vstupního řetězce, a simuluje se tak tvorbu derivačního stromu od kořene k listům, zatímco v případě analýzy zdola nahoru se provádí sestavení derivačního stromu od listů ke kořeni. V praxi to znamená, že je používit pravý rozbor, tj. reverzovaná posloupnost pravidel používající nejpravější derivaci vstupního řětězce. Pro oba přístupy existuje dále více variant, z nichž byla užita tzv. prediktivní syntakticá analýza pro analýzu shora dolů. Tato metoda vyžaduje několik mezikroků vč. vytvoření tzv. LL tabulky, ale následná analýza je univerzální a nezávislá na konkrétní tabulce, což může být velmi užitečné. Pro analýzu zdola nahoru byla naopak použita jednodušší ze dvou variant, a to precedenční syntaktický analyzátor. Tato metoda je opět založena na sestavené speciální tabulce definující precedenci operátorů, závorek, identifikátorů a dalších symbolů a je velmi vhodná pro analýzu matematických výrazů. Tímto přístupem by se složitě popisovala struktura zdrojového kódu, ale díky spojení těchto dvou metod lze provést syntakticou analýzu celého vstupního textu pomocí dvou jednodušších gramatik, což je jedna z hlavních myšlenek gramatických systémů a i jejich analogií, které zobrazují situace, kdy se více účastníků podílí na řešení společného problému a každý přispěje svými znalosti. Samostatně by celý problém nevyřešili, ale společně je to možné.

V následujících kapitolách jsou popsány oba přístupy, které byly použity ve výsledné aplikaci a porovnání s jejich alternativami. Samotný závěr práce je věnován implementaci a ukázce výsledné aplikace.

6.1 LL gramatiky bez epsilon pravidel

Jednou z hlavních překážek využití bezkontextových gramatik pro syntaktickou analýzu shora dolů je existence nejednoznačnosti, která umožňuje vytvoření více derivačních stromů pro stejnou větnou formu. Tato skutečnost vnáší do celého přístupu nedeterminističnost, což je při praktickém použití problém. Vznikl tak speciální typ gramatik, které nesou označení LL gramatiky. Název vznikl ze způsobu analýzy, která probíhá zleva doprava (***Left***) a konstruuje se nejlevější derivace věty (***Leftmost***). Nejdříve bude vysvětlena základní definice LL gramatiky bez epsilon pravidel a poté postupně popsáno vytvoření pomocných množin s následnou konstrukcí LL tabulky, která je nezbytná pro vytvoření prediktivního syntaktického analyzátoru. Definice LL gramatiky bez epsilon pravidel je následující:

Nechť $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ je bezkontextová gramatika bez ε-pravidel. $\boldsymbol{G}$ je LL gramatika, pokud pro každé $\boldsymbol{a} \in \boldsymbol{T}$ a $\boldsymbol{A} \in \boldsymbol{N}$ existuje maximálně jedno pravidlo tvaru $\boldsymbol{A} \rightarrow \boldsymbol{X_1 X_2 \dots X_n} \in \boldsymbol{P}$ a zároveň platí, že $\boldsymbol{a} \in \boldsymbol{First}(\boldsymbol{X_1 X_2 \dots X_n})$.

6.1.1 Množina First

LL gramatika bez epsilon pravidel je poměrně jednoduchá na použití, a proto bude vhodné na ní vysvětlit hlavní principy a poté pokračovat komplexnější verzí s epsilon pravidly. Hlavní myšlenkou množiny $\boldsymbol{First}$ a zároveň i algoritmu $\boldsymbol{First}(\boldsymbol{x})$ je výpočet včech terminálních symbolů, kterými může začínat větná forma derivovaná z $\boldsymbol{x}$. Zjednodušeně řečeno, pokud je provedena derivace $\boldsymbol{x}$, tak všechny terminální symboly, které se mohou objevit na prvním místě místo $\boldsymbol{x}$, musí být vloženy do množiny $\boldsymbol{First}(\boldsymbol{x})$. Pro terminální symboly bude jejich množina $\boldsymbol{First}$ obsahovat sebe sama, zatímco pro neterminály je nutné určit všechny možné terminály, které se z nich mohou derivovat. Tímto se daný algoritmus stává složitějším i přesto, že dosud nebyla použita epsilon pravidla, se kterými je výpočet ještě komplikovanější. Formální definice množiny $\boldsymbol{First}$ je následující:

Nechť $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ je bezkontextová gramatika. Pro každé $\boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T})^*$ je definována množina $\boldsymbol{First}(\boldsymbol{x})$ jako: $\boldsymbol{First}(\boldsymbol{x}) = \{\boldsymbol{a}: \boldsymbol{a} \in \boldsymbol{T}, \boldsymbol{x} \Rightarrow^* \boldsymbol{ay}, \boldsymbol{y} \in (\boldsymbol{N} \cup \boldsymbol{T})^* \}$.

Algoritmus pro výpočet $\boldsymbol{First}(\boldsymbol{x})$ očekává na vstupu bezkontextovou gramatiku bez epsilon pravidel a jeho výstupem je množina $\boldsymbol{First}(\boldsymbol{x})$ pro každý terminál i neterminál.

1. Každému terminálu $\boldsymbol{a} \in \boldsymbol{T}$ vytvoř množinu $\boldsymbol{First}(\boldsymbol{a}) = \{\boldsymbol{a}\}$
2. Aplikuj následující pravidlo, dokud bude možné měnit nějakou množinu $\boldsymbol{First}$:
   if $\boldsymbol{A} \rightarrow \boldsymbol{X_1 X_2} \dots \boldsymbol{X_n} \in \boldsymbol{P}$ **then** přiděj $\boldsymbol{First}(\boldsymbol{X_1})$ do $\boldsymbol{First}(\boldsymbol{A})$

Příklad 1.: Mějme LL gramatiku pro jednoduchý program s následujícími pravidly:

```
1: <prog>   →   begin <state>
2: <state>  →   <cmd> <item> ; <state>
3: <state>  →   end
4: <cmd>    →   print
5: <cmd>    →   read
6: <item>   →   int
7: <item>   →   id
```

Jak lze vidět z jednotlivých pravidel, tento jazyk vyžaduje, aby program začal klíčovým slovem `begin` a skončil slovem `end`, mezi kterými může být libovolné množství příkazů `print` a `read`, které mají oba jako parametr buďto číslo (`int`) nebo proměnnou označenou jako `id`. Nejdříve se sestaví množina $\boldsymbol{First}$ pro všechny terminály a neterminály, a poté je již možné zkonstruovat LL tabulku.

Množiny $\boldsymbol{First}$ pro všechny terminály dle kroku 1.:

```
First(begin) = {begin}
First(;) = {;}
First(print) = {print}
First(read) = {read}
First(end) = {end}
First(int) = {int}
First(id) = {id}
```

Výpočet množin $\boldsymbol{First}$ pro všechny neterminály dle kroku 2.:

<prog> → begin ∈ P: přidej First(begin) do First(<prog>)

First(<prog>) = {begin}

<state> → <cmd> ∈ P: přidej First(<cmd>) do First(<state>)

<state> → end ∈ P: přidej First(end) do First(<state>)

First(<state>) = {end}

First(<cmd>) zatím nebylo zpracováno, takže se z ní do First(<state>) nic nepřidalo a bude nutné se v další iteraci k této množině pravděpodobně vrátit a doplnit ji. Tuto situaci ošetřuje podmínka druhého pravidla, která požaduje opakované iterování, dokud je možné něco měnit.

<cmd> → print ∈ P: přidej First(print) do First(<cmd>)

<cmd> → read ∈ P: přidej First(read) do First(<cmd>)

First(<cmd>) = {print, read}

<item> → int ∈ P: přidej First(int) do First(<item>)

<item> → id ∈ P: přidej First(id) do First(<item>)

First(<item>) = {int, id}

Nyní se provede druhá iteraci, ve které lze změnit už jenom výše zmíněnou množinu First(<state>).

<state> → <cmd> ∈ P: přidej First(<cmd>) do First(<state>)

First(<state>) = {end, print, read}

Tímto jsou zpracovány všechny množiny $\boldsymbol{First}$**,** žádná z nich již nejde změnit, takže algoritmus končí a následuje konstrukce LL tabulky. LL tabulka je velmi užitečný nástroj, který jednoznačně určuje, jaké pravidlo aplikovat na základě aktuálního neterminálu a terminálu (tokenu) na vstupu. Pokud se daná kombinace v tabulce nevyskytuje, nastává syntaktická chyba.

$\boldsymbol{\alpha}$	…	$\boldsymbol{a}$	…
…			
$\boldsymbol{A}$		$\boldsymbol{\alpha(A, a)}$	
…			

Tabulka 1: Ukázka rozložení LL tabulky

Jak je možné vidět v tabulce, řádky indexují neterminály a sloupce terminály. Každá buňka pak obsahuje číslo pravidla, které se má aplikovat pro danou kombinaci. Formálně je definována následovně:

$\boldsymbol{\alpha(A, a) = A \rightarrow X_1 X_2 \ldots X_n \in P}$ pokud $\boldsymbol{a \in First(X_1)}$,

jinak je $\boldsymbol{\alpha(A, a)}$ prázdné => syntaktická chyba.

	begin	;	end	print	read	int	id
<prog>	1						
<state>			3	2	2		
<cmd>				4	5		
<item>						6	7

Tabulka 2: LL tabulka pro Příklad 1

6.2 LL gramatiky s epsilon pravidly

Pro obecnou představu je základní myšlenka konstrukce LL tabulky dostačující, ale pro využití systému v praxi by byly LL gramatiky bez epsilon pravidel velmi omezující, protože z reálného prostředí by popsaly jen malé množství systémů. Je proto nutné zavést LL gramatiky s epsilon pravidly. Jediné, v čem se liší od LL gramatik bez epsilon pravidel, je složitost sestavení LL tabulky. Ta u gramatik s epsilon pravidly vyžaduje několik pomocných množin, které navzájem využívají svoje data, a LL tabulka se sestaví pomocí poslední vytvořené množiny, takzvané množiny $\boldsymbol{Predict}$. Nyní bude uvedena obecná definice LL gramatik s epsilon pravidly a poté představeny jednotlivé pomocné množiny. K ukázce poslouží trochu upravený příklad z předchozí části. Postupně budou sestrojeny dané množiny, až opět vznikne LL tabulka. Následně bude uveden příklad jejího použití při syntaktické analýze, která se pro sestavenou LL tabulku neliší, zda se jednalo o gramatiku s epsilon pravidly nebo bez.

Nechť $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ je bezkontextová gramatika. $\boldsymbol{G}$ je LL gramatika s epsilon pravidly, pokud pro každé $\boldsymbol{a} \in \boldsymbol{T}$ a $\boldsymbol{A} \in \boldsymbol{N}$ existuje maximálně jedno pravidlo tvaru $\boldsymbol{A} \rightarrow \boldsymbol{X_1 X_2 \dots X_n} \in \boldsymbol{P}$ a zároveň platí, že $\boldsymbol{a} \in \boldsymbol{Predict}(\boldsymbol{X_1 X_2 \dots X_n})$.

V definici se objevuje množina $\boldsymbol{Predict}$, která je právě tou poslední množinou při vytváření LL tabulky. Kromě již známé množiny $\boldsymbol{First}$ je potřeba se seznámit ještě s dalšími dvěma množinami, a to $\boldsymbol{Empty}$ a $\boldsymbol{Follow}$. Základním problémem LL gramatiky s epsilon pravidly je jistá neurčitost terminálů z množiny $\boldsymbol{First}$. Zatímco v předchozím případě bylo možné se spolehnout, že daný terminál v množině $\boldsymbol{First}$ bude vždy, pokud byl pro tuto množinu vypočítán, tak nyní vzniká problém, že se tento terminál v konkrétní množině $\boldsymbol{First}$ <u>může</u> vyskytovat. Pokud totiž dojde k expanzi neterminálu pomocí pravidla, které obsahuje na pravé straně další neterminály a některé z nich (nebo všechny) derivují epsilon, pak může dojít k jejich vymazání a počáteční terminál může být symbol derivovaný z následujícího neterminálu v pravidle nebo může vypadnout celý, na počátku přepisovaný neterminál, a prvním terminálem v tomto místě bude až první terminál za tímto neterminálem.

Mějme pravidlo $\boldsymbol{r}$: $\boldsymbol{A} \rightarrow \boldsymbol{X_1 X_2 \dots X_n} \in \boldsymbol{P}$ a aktuální větnou formu $\boldsymbol{xAy}$. V případě, že by se jednalo o gramatiku bez epsilon pravidel, tak po derivaci $\boldsymbol{A}$ pomocí pravidla $\boldsymbol{r}$ by se jako první terminál ve větné formě objevilo místo $\boldsymbol{A}$ nějaké $\boldsymbol{a} \in \boldsymbol{First}(\boldsymbol{A})$, které by zároveň bylo ve $\boldsymbol{First}(\boldsymbol{X_1})$ atd. Jenomže pokud $\boldsymbol{X_1 X_2 \dots X_n}$ může derivovat epsilon, pak mohou některá vypadnout a může dojít k situaci, kdy buď $\boldsymbol{a} \in \boldsymbol{First}(\boldsymbol{X_i})$ pro $2 \leq \boldsymbol{i} \leq \boldsymbol{n}$, nebo mohou být vymazána všechna, pak $\boldsymbol{a} \in \boldsymbol{First}(\boldsymbol{y})$.

Pro epsilon pravidla je výpočet mnohem složitější a je nutné prozkoumat gramatiku opravdu do hloubky, odhalit všechny možnosti a kombinace, které symboly mohou být kterými nahrazeny a

zastoupeny. K tomu pomůže postupné a systematické sestavení všech čtyř množin $\boldsymbol{Empty}$, $\boldsymbol{First}$, $\boldsymbol{Follow}$ a $\boldsymbol{Predict}$.

Pro výpočet množin se zavádí pomocný symbol $, který slouží k zakončemí každého vstupního řetězece. Jedná se o nutnost způsobenou epsilon pravidly. Pokud by se celý vstupní řetězec derivoval na epsilon, je jisté, že symbol $ zůstane, a definuje tak konec vstupu.

6.2.1 Množina Empty

Množina $\boldsymbol{Empty}$ je nejjednodušší a pomáhá určit, jestli daný symbol derivuje epsilon nebo ne. Je to tedy startující bod pro analýzu epsilon derivací, které nemusí být vidět v gramatice na první pohled, ale díky návaznosti pravidel může být daný neterminál nahrazen za epsilon i přes několik pravidel (derivací). Obecně je tedy $\boldsymbol{Empty(x)}$ množina, která obsahuje pouze jediný prvek, a to $\boldsymbol{\varepsilon}$, pokud $\boldsymbol{x}$ derivuje $\boldsymbol{\varepsilon}$, jinak je prázdná. Lze tak říci, že množina $\boldsymbol{Empty}$ není přímo množinou tak, jak je běžně chápána, ale pouze dvoustavovým atributem určujícím, jestli se může na místě daného symbolu objevit epsilon nebo ne. Ostatní množiny již obsahují více symbolů, a proto i $\boldsymbol{Empty(x)}$ se považuje za množinu pro jednotný přístup a manipulaci se všemi čtyřmi množinami. Formální definice množiny $\boldsymbol{Empty}$ je následující:

Nechť $\boldsymbol{G = (N, T, P, S)}$ je bezkontextová gramatika. Pro každé $\boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T})^*$ je definována množina $\boldsymbol{Empty(x)}$ jako: $\boldsymbol{Empty(x)} = \{\boldsymbol{\varepsilon}\}$ pokud $\boldsymbol{x} \Rightarrow^* \boldsymbol{\varepsilon}$ jinak $\boldsymbol{Empty(x)} = \emptyset$.

Algoritmus pro výpočet $\boldsymbol{Empty(x)}$ očekává na vstupu bezkontextovou gramatiku a jeho výstupem je množina $\boldsymbol{Empty(x)}$ pro každý terminál i neterminál.

1. Každému terminálu $\boldsymbol{a} \in \boldsymbol{T}$ vytvoř množinu $\boldsymbol{Empty(a)} = \emptyset$
2. Pro každý neterminál $\boldsymbol{A} \in \boldsymbol{N}$ rozhodni:
   pokud $\boldsymbol{A} \rightarrow \boldsymbol{\varepsilon} \in \boldsymbol{P}$ pak $\boldsymbol{Empty(A)} = \{\boldsymbol{\varepsilon}\}$, jinak $\boldsymbol{Empty(A)} = \emptyset$.
3. Aplikuj následující pravidlo, dokud bude možné měni nějakou množinu $\boldsymbol{Empty}$:
   pokud $\boldsymbol{A} \rightarrow \boldsymbol{X_1 X_2} \dots \boldsymbol{X_n} \in \boldsymbol{P}$ a zároveň $\boldsymbol{Empty(X_i)} = \{\boldsymbol{\varepsilon}\}$ pro všechna $\boldsymbol{i = 1, .., n}$,
   pak $\boldsymbol{Empty(A)} = \{\boldsymbol{\varepsilon}\}$

Příklad 2.: Mějme LL gramatiku s epsilon pravidly pro program s následujícími pravidly:

```
1: <prog>   →   begin <state>
2: <state>  →   <cmd> <item> ; <state>
3: <state>  →   end
4: <cmd>    →   print
5: <cmd>    →   read
6: <cmd>    →   ε
7: <item>   →   int
8: <item>   →   id
```

Pro praktickou ukázku je mírně upraven příklad z předchozí části pouze přidáním jednoho pravidla, pravidla č. 6, které umožňuje, aby neterminál `<cmd>` derivoval epsilon. Mohlo by se zdát, že se jedná o triviální změnu, která nemůže mít zásadní vliv na konstrukci LL tabulky, ale opak je pravdou. Může

zde dojít ke vzorové situaci, která byla popsána na začátku této kapitoly o LL gramatikách s epsilon pravidly. Pokud se totiž v pravidle č. 2 derivuje první neterminál `<cmd>` na epsilon, pak pro $\boldsymbol{First}(<\textbf{state}>)$ nemusí být první terminál ve $\boldsymbol{First}(<\textbf{cmd}>)$, ale až ve $\boldsymbol{First}(<\textbf{item}>)$. Pokud by byla gramatika složitější a existovalo pravidlo `<item> → ε`, potom by mohl být prvním terminálem až symbol '**;**' atd. Je tedy nutné při výpočtu LL tabulky zohlednit tuto neurčitost a zahrnout do množin $\boldsymbol{First}$ všechny potencionální možnosti, když se někde objeví epsilon.

Množiny $\boldsymbol{Empty}$ pro všechny terminály dle kroku 1.:

```
Empty(begin) = ∅
Empty(;) = ∅
Empty(print) ∅
Empty(read) = ∅
Empty(end) = ∅
Empty(int) = ∅
Empty(id) = ∅
```

Výpočet množin $\boldsymbol{First}$ pro všechny neterminály dle kroku 2.:

```
<prog> → begin ∈ P:          <prog> → ε ∉ P
Empty(<prog>) = ∅

<state> → <cmd> ∈ P:
<state> → end ∈ P:           <state> → ε ∉ P
Empty(<state>) = ∅

<cmd> → print ∈ P:
<cmd> → read ∈ P:
<cmd> → ε ∈ P:               <cmd> → ε ∈ P
Empty(<cmd>) = {ε}

<item> → int ∈ P:
<item> → id ∈ P:             <item> → ε ∉ P
Empty(<item>) = ∅
```

Po prvním průchodu přes neterminály gramatiky je patrné, že pouze pro neterminál `<cmd>` je množina $\boldsymbol{Empty}(<\boldsymbol{cmd}>) = \{\boldsymbol{\varepsilon}\}$, ostatní neterminály epsilon přímo nederivují. Při provedení dalšího průchodu pro splnění kroku č. 3 algoritmu $\boldsymbol{Empty}$, je zřejmé, že se již žádná množina $\boldsymbol{Empty}$ nezmění. Jediným kandidátem na změnu by byla množina $\boldsymbol{Empty}(<\boldsymbol{state}>)$, protože pravidlo č. 2 obsahuje neterminál `<cmd>`, jehož množina $\boldsymbol{Empty}$ ve druhém průchodu již obsahuje $\boldsymbol{\varepsilon}$, ale pravá strana tohoto pravidla obsahuje další neprázdné neterminály a zároveň i jeden terminál, takže neplatí podmínka pro pravou stranu pravidla $\boldsymbol{Empty}(\boldsymbol{X_i}) = \{\boldsymbol{\varepsilon}\}$ pro všechna $\boldsymbol{i} = \mathbf{1}, .., \boldsymbol{n}$, a žádná množina $\boldsymbol{Empty}$ již opravdu nejde změnit. Tímto byly vyřešeny všechny množiny $\boldsymbol{Empty}$ pro danou gramatiku a následuje modifikovaná verze množiny $\boldsymbol{First}$ pro LL gramatiky s epsilon pravidly.

6.2.2 Množina First

Význam a základní myšlenka množiny $\boldsymbol{First}$ byla uvedena již v předchozí sekci o LL gramatikách bez epsilon pravidel, takže není nutné zacházet do takové hloubky a lze přejít rovnou ke změněnému algoritmu. Největším problémem epsilon pravidel je, jak již bylo zmíněno, nutnost ošetření případu, kdy dojde ke smazání prvních neterminálů (jednoho či více) pravé strany pravidla (přepsáním na epsilon), a následující terminály a neterminály se tak posunou na jejich místo. Za normálních okolností by se vyskytovaly až za nimi. Musí se tedy do množin $\boldsymbol{First}$ zahrnout nejen terminály, které se mohou derivovat na místě prvního neterminálu na pravé straně jednotlivých pravidel, ale postihnout i případy, kdy je tento první neterminál smazán. Potom se musí zahrnout do dané množiny $\boldsymbol{First}$ terminály následujícího neterminálu. Může nastat i situace, že se tímto způsobem dojde až na konec pravé strany pravidla, pak je potřeba sjednotit všechny množiny $\boldsymbol{First}$ jednotlivých neterminálů vyskytujících se na pravé straně tohoto pravidla, protože nikdy není dopředu zřejmé, kolik z nich může být zleva popořadě přepsáno na epsilon. Situaci, kdy by byly přepsány na epsilon všechny (množina $\boldsymbol{Empy}$ neterminálu na levé straně pravidla by obsahovala epsilon), řeší později množina $\boldsymbol{Follow}$.

Algoritmus pro výpočet $\boldsymbol{First(x)}$ očekává na vstupu bezkontextovou gramatiku a jeho výstupem je množina $\boldsymbol{First(x)}$ pro každý terminál i neterminál.

1. Každému terminálu $\boldsymbol{a} \in \boldsymbol{T}$ vytvoř množinu $\boldsymbol{First(a)} = \{\boldsymbol{a}\}$
2. Každému neterminálu $\boldsymbol{A} \in \boldsymbol{N}$ vytvoř množinu $\boldsymbol{First(A)} = \emptyset$
3. Aplikuj následující pravidlo, dokud bude možné měnit nějakou množinu $\boldsymbol{First}$:
   pokud $\boldsymbol{A \rightarrow X_1X_2 \dots X_{k-1}X_k \dots X_n \in P}$
    a. přidej všechny symboly z $\boldsymbol{First(X_1)}$ do $\boldsymbol{First(A)}$
    b. pokud $\boldsymbol{Empty(X_i)} = \{\boldsymbol{\varepsilon}\}$ pro $\boldsymbol{i = 1, .., k_{-1}, k \leq n}$, přidej všechny symboly z $\boldsymbol{First(X_k)}$ do $\boldsymbol{First(A)}$

Algoritmus je velmi podobný původní variantě, pouze s tím rozdílem, že předtím se automaticky přiřadily $\boldsymbol{First(X_1)}$ do $\boldsymbol{First(A)}$, což se nyní stane také, ale zároveň se kontrolují všechna $\boldsymbol{X_i}$ zleva doprava a pokud tvoří skevenci, kde jednotlivá $\boldsymbol{Empty(X_i)} = \{\boldsymbol{\varepsilon}\}$, pak jsou postupně přidávána $\boldsymbol{First(X_i)}$ do $\boldsymbol{First(A)}$, dokud podmínka nepřestane platit nebo se nedojde na konec pravé strany pravidla. Vždy je přidána i první množna $\boldsymbol{First(X_i)}$ pro $\boldsymbol{X_i}$, u kterého již neplatí $\boldsymbol{Empty(X_i)} = \{\boldsymbol{\varepsilon}\}$, popř. $\boldsymbol{X_n}$, pokud $\boldsymbol{X_n = X_k}$.

Množiny $\boldsymbol{First}$ pro všechny terminály (1.):

```
First(begin) = {begin}
First(;) = {;}
First(print) = {print}
First(read) = {read}
First(end) = {end}
First(int) = {int}
First(id) = {id}
```

Množiny $\boldsymbol{First}$ pro všechny neterminály (2.):

```
First(<prog>) = ∅
First(<state>) = ∅
First(<cmd>) = ∅
First(<item>) = ∅
```

Po inicializaci všech množin ***First*** je možné pokračovat jejich výpočtem dle kroku 3, ke kterému jsou potřeba množiny ***Empty*** z předchozí kapitoly:

```
<prog> → begin <state> ∈ P:
      Empty(begin)≠{ε}
            přidej First(begin)   do First(<prog>)
First(<prog>) = {begin}

<state> → <cmd> <item> ; <state> ∈ P:
      Empty(<cmd>)={ε}
            přidej First(<cmd>)   do First(<state>)
      Empty(<item>)≠{ε}
            přidej First(<item>)  do First(<state>)
<state> → end ∈ P:
      Empty(end)≠{ε}
            přidej First(end)     do First(<state>)
First(<state>) = {end}

<cmd> → print ∈ P:
      Empty(print)≠{ε}
            přidej First(print)   do First(<cmd>)
<cmd> → read ∈ P:
      Empty(read)≠{ε}
            přidej First(read)    do First(<cmd>)
First(<cmd>) = {print, read}

<item> → int ∈ P:
      Empty(int)≠{ε}
            přidej First(int)     do First(<item>)
<item> → id ∈ P:
      Empty(id)≠{ε}
            přidej First(id)      do First(<item>)
First(<item>) = {int, id}
```

Nyní se provede druhá iterace, ve které lze změnit opět už jenom množinu `First(<state>)`, nyní však na ni bude mít vliv jak množina `First(<cmd>)`, tak i změněná množina `First(<item>)` :

```
<state> → <cmd> <item> ; <state> ∈ P:
      Empty(<cmd>)={ε}
            přidej First(<cmd>)   do First(<state>)
      Empty(<item>)≠{ε}
            přidej First(<item>)  do First(<state>)
First(<state>) = {end, print, read, int, id}
```

Lze zde vidět zásadní rozdíl oproti předchozí ukázce bez epsilon pravidel. Množina `First(<state>)` se rozšířila o dva terminály `int` a `id` díky tomu, že v pravidle č. 2 může být první neterminál `<cmd>` smazán pomocí epsilon pravidla a na jeho místo se tak dostane druhý neterminál `<item>`.

6.2.3 Algoritmus First($X_1X_2 \ldots X_n$)

Pro výpočet zbývajících množin se zavádí rozšíření stávajících algoritmů $\boldsymbol{Empty}(\boldsymbol{x})$ **a** $\boldsymbol{First}(\boldsymbol{x})$, aby dokázaly vypočítat příslušné množiny pro celé pravé strany pravidel, takže ne jenom pro jeden symbol $\boldsymbol{x}$, ale pro celou sekvenci symbolů $\boldsymbol{X_1X_2 \ldots X_n}$.

Algoritmus pro výpočet $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_n})$ očekává na vstupu bezkontextovou gramatiku $G = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ a předpočítané množiny $\boldsymbol{First}(\boldsymbol{X})$ a $\boldsymbol{Empty}(\boldsymbol{X})$ pro každé $\boldsymbol{X} \in \boldsymbol{N} \cup \boldsymbol{T}$, $\boldsymbol{x} = \boldsymbol{X_1X_2 \ldots X_n}$, kde $\boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T})^+$. Výstupem je potom množina $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_n})$. Dále platí, že $\boldsymbol{First}(\boldsymbol{\varepsilon}) = \emptyset$.

1. $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_n}) = \boldsymbol{First}(\boldsymbol{X_1})$
2. Aplikuj následující pravidlo, dokud bude možné měnit množinu $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_{k-1}X_k \ldots X_n})$:
   pokud $\boldsymbol{Empty}(\boldsymbol{X_i}) = \{\boldsymbol{\varepsilon}\}$ pro $\boldsymbol{i} = \boldsymbol{1}, .., \boldsymbol{k_{-1}}, \boldsymbol{k} \leq \boldsymbol{n}$, přidej všechny symboly z $\boldsymbol{First}(\boldsymbol{X_k})$ do $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_{k-1}X_k \ldots X_n})$

Algoritmus se příliš neliší od předchozí verze. Opět se v prvním kroku vypořádá s případem bez epsilon pravidel, kdy do výsledné množiny $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_n})$ přiřadí $\boldsymbol{First}(\boldsymbol{X_1})$, a poté se již opakuje situace z předchozího algoritmu, kde postupně testuje množiny $\boldsymbol{Empty}$ pro jednotlivá $\boldsymbol{X_i}$, a přidává jejich množiny $\boldsymbol{First}$ do $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_n})$, dokud nenarazí na první, u které již neplatí, že se její $\boldsymbol{Empty}(\boldsymbol{X_i}) = \{\boldsymbol{\varepsilon}\}$. Její $\boldsymbol{First}$ ještě přidá do $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_n})$ a pokud už nemůže nic měnit, výpočet končí.

6.2.4 Algoritmus Empty($X_1X_2 \ldots X_n$)

Rozšíření $\boldsymbol{Empty}(\boldsymbol{X_1X_2 \ldots X_n})$ bylo zavedeno ze stejného důvodu jako předchozí algoritmus pro $\boldsymbol{First}(\boldsymbol{X_1X_2 \ldots X_n})$. Bude potřeba u následujících množin a jeho výpočet je ze všech nejjednodušší. Umožňuje určit, jestli může být celá pravá strana pravidla prázdná ($\boldsymbol{Empty}$) nebo se z ní vždy nějaký terminál vygeneruje.

Algoritmus pro výpočet $\boldsymbol{Empty}(\boldsymbol{X_1X_2 \ldots X_n})$ očekává na vstupu bezkontextovou gramatiku $G = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ a předpočítanou množinu $\boldsymbol{Empty}(\boldsymbol{X})$ pro každé $\boldsymbol{X} \in \boldsymbol{N} \cup \boldsymbol{T}$, $\boldsymbol{x} = \boldsymbol{X_1X_2 \ldots X_n}$, kde $\boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T})^+$. Výstupem je potom množina $\boldsymbol{Empty}(\boldsymbol{X_1X_2 \ldots X_n})$. Dále platí, že $\boldsymbol{Empty}(\boldsymbol{\varepsilon}) = \{\boldsymbol{\varepsilon}\}$.

1. Pokud $\boldsymbol{Empty}(\boldsymbol{X_i}) = \{\boldsymbol{\varepsilon}\}$ pro $\boldsymbol{i} = \boldsymbol{1}, .., \boldsymbol{n}$, potom $\boldsymbol{Empty}(\boldsymbol{X_1X_2 \ldots X_n}) = \{\boldsymbol{\varepsilon}\}$
2. jinak $\boldsymbol{Empty}(\boldsymbol{X_1X_2 \ldots X_n}) = \emptyset$

6.2.5 Množina Follow

Množina $\boldsymbol{Follow}$ umožňuje pokrýt ty případy, kdy se v dané větné formě díky epsilon pravidlu smaže celý neterminál a je potřeba předem vědět, co za ním může následovat, resp. se dostat na jeho místo. $\boldsymbol{Follow(A)}$ je tedy množina všech terminálů, které se mohou vyskytovat vpravo bezprostředně od $\boldsymbol{A}$ ve větné formě.

Nechť $\boldsymbol{G = (N, T, P, S)}$ je bezkontextová gramatika. Pro všechna $\boldsymbol{A \in N}$ je definována množina $\boldsymbol{Follow(A)}$ jako:

$$\boldsymbol{Follow(A)} = \{\boldsymbol{a} : \boldsymbol{a} \in \mathbf{T}, \boldsymbol{S} \Rightarrow^* \mathbf{xAay}, \mathbf{x}, \mathbf{y} \in (\mathbf{N} \cup \mathbf{T})^* \} \cup \{\$: \boldsymbol{S} \Rightarrow^* \boldsymbol{xA}, \boldsymbol{x} \in (\mathbf{N} \cup \mathbf{T})^*\}$$

Musí se tak pro všechny neterminály vypočítat jejich množiny $\boldsymbol{Follow}$, tedy určit všechny možné terminály, které za nimi mohou následovat pro případ, že by byl daný neterminál smazán pomocí epsilon pravidla.

Algoritmus pro výpočet $\boldsymbol{Follow(A)}$ očekává na vstupu bezkontextovou gramatiku $G = \boldsymbol{(N, T, P, S)}$ a předpočítané množinu $\boldsymbol{Empty(X)}$ a $\boldsymbol{First(X)}$. Výstupem je potom množina $\boldsymbol{Follow(A)}$ pro každé $\boldsymbol{A \in N}$.

1. $\boldsymbol{Follow(S)} = \{\$\}$
2. Aplikuj následující pravidlo, dokud bude možné měnit nějakou množinu $\boldsymbol{Follow}$:
   pokud $\boldsymbol{A \rightarrow xBy \in P}$ potom
   a. Pokud $\boldsymbol{y \neq \varepsilon}$ potom přidej všechny symboly z $\boldsymbol{First(y)}$ do $\boldsymbol{Follow(B)}$
   b. Pokud $\boldsymbol{Empty(y)} = \{\boldsymbol{\varepsilon}\}$ potom přidej všechny symboly z $\boldsymbol{Follow(A)}$ do $\boldsymbol{Follow(B)}$

Objevuje se zde s již dříve zmíněný symbolem \$, který pomáhá určit konec vstupního řetězce, a proto se přiřazuje hned na začátku algoritmu do $\boldsymbol{Follow(S)}$. To je vlastně nejvýstižnější popis množiny $\boldsymbol{Follow}$. Předpokládá se, že je vstupní řetezec vždy zakončen symbolem \$. Pokud by byl celý vstupní řetězec prázdný, tedy existovalo by např. pravidlo $\boldsymbol{S \rightarrow \varepsilon}$, pak bezprostředně za celým řetězcem, který se může derivovat z $\boldsymbol{S}$ (např. celý vstupní program), vždy následuje symbol \$. V tomto případě dokonce místo neterminálu $\boldsymbol{S}$, protože byl epsilon pravidlem smazán. Bod a) druhého kroku postihuje variantu, kdy se za neterminálem $\boldsymbol{B}$ vyskytují další symboly, potom se přidá jejich množina $\boldsymbol{First}$ do $\boldsymbol{Follow(B)}$. Druhá nezávislá podmínka v bodě b) ověří, jestli mohou být symboly za neterminálem $\boldsymbol{B}$ $\boldsymbol{Empty}$. Buďto mohou být smazány epsilon pravidlem nebo se jedná o první neterminál zprava, v tom případě platí dodatek $\boldsymbol{Empty(\varepsilon)} = \{\boldsymbol{\varepsilon}\}$. V obou případech je přidána množina $\boldsymbol{Follow(A)}$ do $\boldsymbol{Follow(B)}$, aby byla k dispozici informace, co může následovat bezprostředně za neterminálem $\boldsymbol{B}$. Tímto způsobem se určí množiny $\boldsymbol{Follow}$ pro všechny neterminály.

Výpočet množin $\boldsymbol{Follow}$ pro krok 1.:

```
Follow(<prog>) = {$}
```

Nyní se bude provádět krok 2. pro všechny neterminály, dokud bude možné měnit některou z množin $\boldsymbol{Follow}$:

```
<prog> →  begin <state> ε
      Empty(ε)={ε}
            přidej Follow(<prog>) do Follow(<state>)
```

Follow(<state>)={$}

```
<state>     →   <cmd> <item> ; <state> ε
      Empty(ε)={ε}
            přidej Follow(<state>) do Follow(<state>)
```

Follow(<state>)={$}

```
<state>     →   <cmd> <item> ; <state>
      ; <state> ≠ ε
            přidej First(; <state>) do Follow(<item>)
      Empty(; <state>)≠{ε}
```

Follow(<item>)={;}

```
<state>     →   <cmd> <item> ; <state>
      <item> ; <state> ≠ ε
            přidej First(<item> ; <state>) do Follow(<cmd>)
      Empty(<item> ; <state>)≠{ε}
```

Follow(<cmd>)={int, id}

6.2.6 Množina Predict

Množina $\boldsymbol{Predict}$ je poslední množinou, která je nutná pro sestavení LL tabulky, resp. ostatní množiny sloužily pouze jako podpora k sestavení množiny $\boldsymbol{Predict}$, ze které se už přímo konstruuje LL tabulka pro LL gramatiku s epsilon pravidly. Množina $\boldsymbol{Predict}$ se váže na jednotlivá pravidla a pro každé pravidlo gramatiky umožňuje $\boldsymbol{Predict}(\boldsymbol{A} \rightarrow \boldsymbol{x})$ určit množinu všech terminálů, které mohou být aktuálně nejlevěji vygenerovány, pokud se pro libovolnou větnou formu použije pravidlo $\boldsymbol{A} \rightarrow \boldsymbol{x}$.

Nechť $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ je bezkontextová gramatika. Pro každé $\boldsymbol{A} \rightarrow \boldsymbol{x} \in \boldsymbol{P}$ je definována množina $\boldsymbol{Predict}(\boldsymbol{A} \rightarrow \boldsymbol{x})$ následovně:

1. Pokud $\boldsymbol{Empty}(\boldsymbol{x}) = \{\boldsymbol{\varepsilon}\}$ potom $\boldsymbol{Predict}\ (\boldsymbol{A} \rightarrow \boldsymbol{x}) = \boldsymbol{First}(\boldsymbol{x}) \cup \boldsymbol{Follow}(\boldsymbol{A})$
2. jinak pokud $\boldsymbol{Empty}(\boldsymbol{x}) = \emptyset$ potom $\boldsymbol{Predict}\ (\boldsymbol{A} \rightarrow \boldsymbol{x}) = \boldsymbol{First}(\boldsymbol{x})$

Nyní se vypočítá množiny $\boldsymbol{Predict}$ pro jednotlivá pravidla gramatiky našeho vzorového příkladu:

```
1: <prog>   →   begin <state>
      Empty(begin <state>) = ∅
      Predict(1) = First(begin <state>) = First(begin) = {begin}

2: <state>  →   <cmd> <item> ; <state>
```

```
    Empty(<cmd> <item> ; <state>) = ∅
    Predict(2) = First(<cmd> <item> ; <state>) =
    = First(<cmd> <item>) = {print, read, int, id}

3: <state>  →   end
    Empty(end) = ∅, Predict(3) = First(end) = {end}

4: <cmd>    →   print
    Empty(print) = ∅, Predict(4) = First(print) = {print}

5: <cmd>    →   read
    Empty(read) = ∅, Predict(5) = First(read) = {read}

6: <cmd>    →   ε
    Empty(ε) = {ε}
    Predict (6) :=  First(ε) ∪ Follow(<cmd>) = {int, id}

7: <item>   →   int
    Empty(int) = ∅, Predict(7) = First(int) = {int}

8: <item>   →   id
    Empty(id) = ∅, Predict(8) = First(id) = {id}
```

Na závěr výpočtu se provede sestavení výsledné LL tabulky pro tento vzorový příklad. K vyplnění tabulky postačí pouze množina $\boldsymbol{Predict}$, ostatní množiny jsou pomocné. Celá tabulka je totožná s předchozí verzí, která již byla detailně popsána v předchozí kapitole. Liší se způsob výběru pravidla pro jednotlivé buňky a přibyl jeden sloupec pro zakončovací terminál $:

$$\boldsymbol{\alpha}(\boldsymbol{A},\boldsymbol{a}) = \boldsymbol{A} \to \boldsymbol{X_1 X_2} \dots \boldsymbol{X_n} \in \boldsymbol{P} \text{ pokud } \boldsymbol{a} \in \boldsymbol{Predict}(\boldsymbol{A} \to \boldsymbol{X_1 X_2} \dots \boldsymbol{X_n}),$$
$$\text{jinak je } \boldsymbol{\alpha}(\boldsymbol{A},\boldsymbol{a}) \text{ prázdné => syntaktická chyba.}$$

	begin	;	end	print	read	int	id	$
<prog>	1							
<state>			3	2	2	2	2	
<cmd>				4	5	6	6	
<item>						7	8	

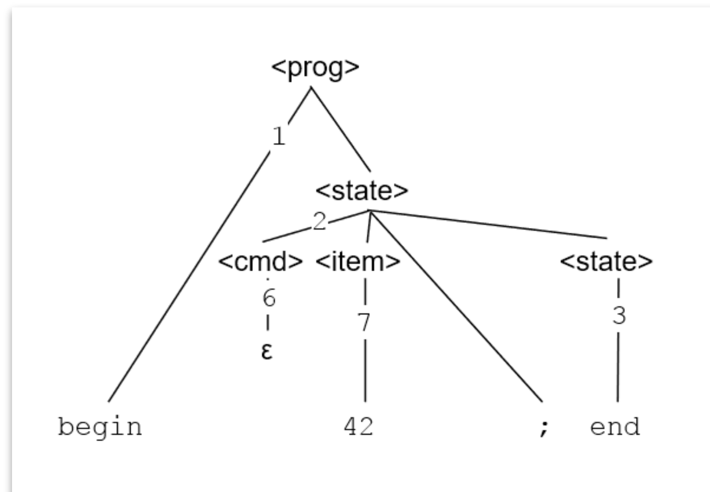
Tabulka 3: Výsledná LL tabulka pro Příklad 2

LL tabulka se oproti předchozí variantě bez epsilon pravidel rozšířila o případy, kdy dojde ke smazání neterminálu `<cmd>`, který má jako jediný epsilon pravidlo, a syntaktický analyzátor tak díky tabulce ví, jak se s touto situací vypořádat, a které symboly očekávat.

Příklad 3.: Proveďte syntaktickou analýzu následujícího vstupního kódu pomocí LL tabulky sestavené výše:

```
begin
   42;
end
```

Následuje ukázka jednoduchého způsobu použití LL tabulky při syntaktické analýze. Zatím pouze intuitivně a velmi triviálně, později bude vysvětlen algoritmus prediktivní syntaktické analýzy, pomocí které lze provádět analýzu automaticky a nad libovolně velkou LL tabulkou.



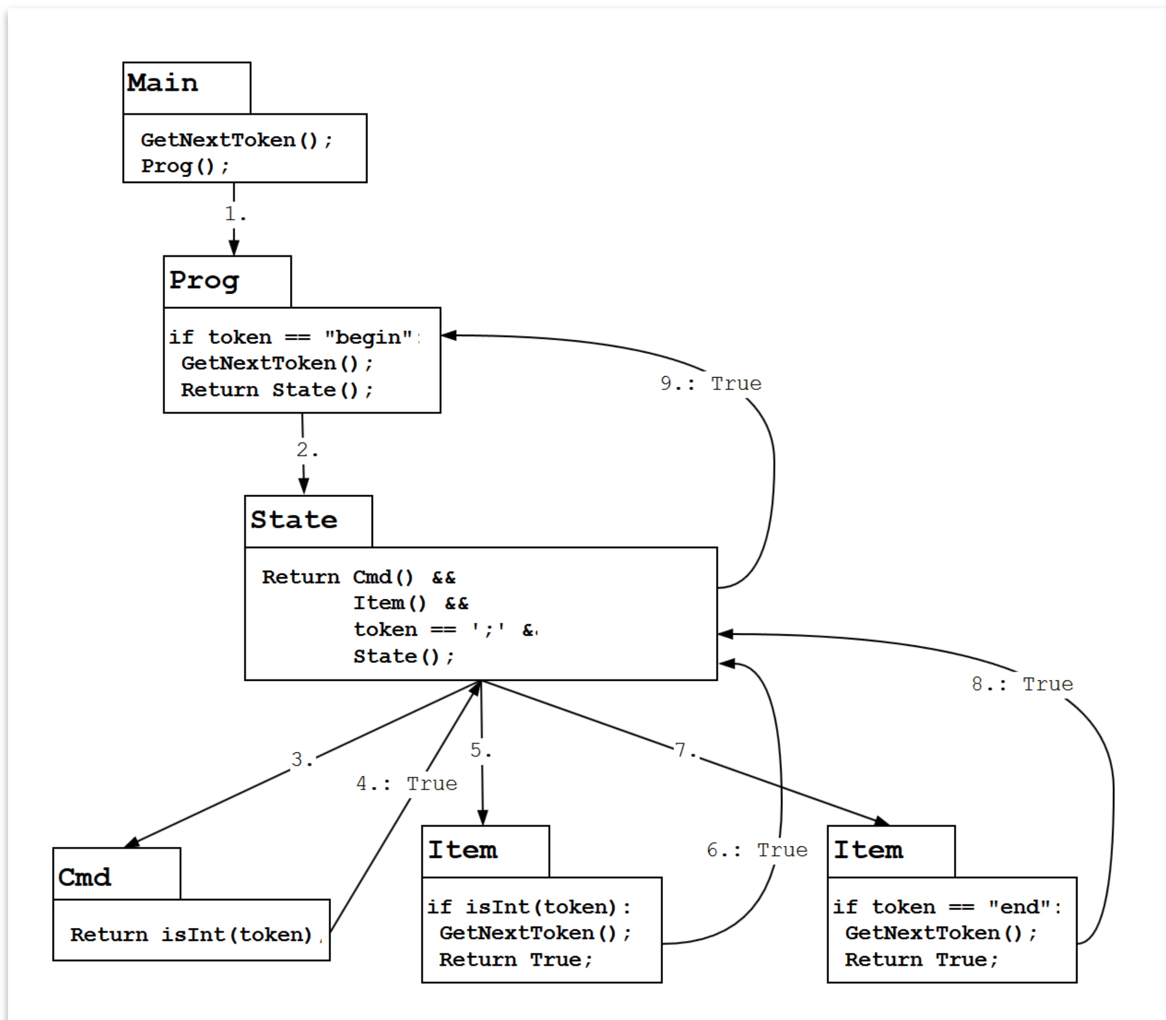
Obrázek 8: Derivační strom příkladu 3 s posloupností aplikovaných pravidel: 12673

6.3 Rekurzivní sestup

Rekurzivní sestup je jednoduchá metoda, která transformuje neterminály gramatiky do odpovídajících procedur, které obstarávají obsluhu jednotlivých akcí, a provádějí tak simulaci pravidel gramatiky. Z programátorského hlediska se jedná o velmi intuitivní přístup. Implemetace rekurzivního sestupu zahrnuje velké množství podmínek a větvení pro kontrolu výskytu jednotlivých terminálů a podle toho se vzájemně rekurzivně volají jednotlivé procedury, které buď odhalí syntaktickou chybu a vrátí logickou hodnotu $\boldsymbol{false}$ nebo zavolají další procedury a postupným zanořováním kontrolují celý vstupní kód, dokud poslední metoda nevrátí $\boldsymbol{true}$, resp. $\boldsymbol{false}$, podle toho, zda byla syntaxe až do konce v pořádku. Výsledná hodnota se poté propaguje jednotlivými úrovněmi zanoření zpět do první metody, která odstartovala syntaktickou analýzu, a ta oznámí uživateli, zda je celý vstup v pořádku nebo se někde vyskytuje syntaktická chyba.

Jak již bylo uvedeno, jedná se o velmi jednoduchou a dobře pochopitelnou metodu, protože programátor implementuje jednotlivá pravidla gramatiky. Pro lepší kontrolu a přehlednost je vhodné opět vycházet z LL tabulky, protože při jejím sestavení se odhalí všechny možnosti a skryté kombinace, které do analýzy vnášejí epsilon pravidla, ale lze sestavit rekurzivní analyzátor i ze samotné gramatiky. Velkou nevýhodou tohoto přístupu je jednoúčelovost použití, protože implementace daného analyzátoru odpovídá konkrétní gramatice, a pokud by bylo třeba později upravit nebo úplně vyměnit gramatiku, musí se celý analyzátor přeprogramovat. Oproti tomu

prediktivní syntaktická analýza využívá při analýze LL tabulku a při změně gramatiky stačí pouze upravit LL tabulku a není potřeba provádět zásah do implementace. Z tohoto důvodu byla v práci upřednostněna implementace prediktivní syntaktické analýzy před rekurzivním sestupem, aby bylo možné využít jejích invariantních vlastností vůči gramatice, i když vždy vyžaduje složitou konstrukci LL tabulky.



Obrázek 9: Ilustrace volání metod rekurzivního sestupu pro vstupní řetězec „`begin 42; end`"

6.4 Prediktivní syntaktická analýza

Prediktivní syntaktický analyzátor je jeden z přístupů syntaktické analýzy shora dolů, který není oproti předchozí variantě rekurzivního sestupu tolik programátorsky intuitivní, ale na druhou stranu celou analýzu řídí jeden obecný algoritmus, který není implementačně závislý na dané gramatice, což přináší velkou výhodu, protože při vhodné implementaci je výsledný analyzátor znovupoužitelný pro různé gramatiky na rozdíl od rekurzivního sestupu.

Prediktivní syntaktický analyzátor se skládá ze svého jádra, které řídí syntaktickou analýzu a čte jednotlivé tokeny zpracované lexikální analýzou. Dále vyžaduje implementaci zásobníku a LL tabulky, která opovídá dané gramatice. Tyto dvě komponenty přidávají prediktivnímu analyzátoru na

složitosti a mohlo by se zdát, že je rekurzivní sestup jednodušší cestou, ale jak již bylo zmíněno, náročnější počáteční implementace se později vrátí ve flexibiliteě a univerzálnosti celého syntaktického analyzátoru.

Syntaktická analýza probíhá shora dolů, takže výstupem analýzy je posloupnost pravidel pro vstupní řetězec, která je použita v nejlevější derivaci. Simuluje se tak sestavení derivačního stromu od kořene k listům.

Algoritmus prediktivní syntaktické analýzy očekává na vstupu LL tabulku pro zvolenou gramatiku $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ a vstupní řetězec tokenů $\boldsymbol{x} \in \boldsymbol{T}^*$. Výstupem je potom levý rozbor pro $\boldsymbol{x}$, pokud je $\boldsymbol{x}$ řetězcem jazyka $\boldsymbol{L}(\boldsymbol{G})$, $\boldsymbol{x} \in \boldsymbol{L}(\boldsymbol{G})$, jinak syntaktický chyba.

1. Ulož symbol $ na zásobník
2. Ulož symbol $\boldsymbol{S}$ na zásobník
3. Nechť $\boldsymbol{X}$ je vrchol zásobníku a $\boldsymbol{a}$ aktuální token
4. Pokud $\boldsymbol{X} = \$$ a zároveň $\boldsymbol{a} = \$$, potom **<u>úspěch</u>**; ukonči algoritmus
5. Pokud $\boldsymbol{X} \in \boldsymbol{T}$ a zároveň $\boldsymbol{X} = \boldsymbol{a}$, potom odeber $\boldsymbol{X}$ z vrcholu zásobníku a načti další $\boldsymbol{a}$ ze vtupního řetězce; vrať se na bod 3
6. Pokud $\boldsymbol{X} \in \boldsymbol{N}$ a zároveň $\boldsymbol{r}: \boldsymbol{X} \rightarrow \boldsymbol{x} \in \boldsymbol{LL} - \boldsymbol{tabulka}[\boldsymbol{X}, \boldsymbol{a}]$, potom zaměň vrchol zásobníku $\boldsymbol{X}$ za řetězec symbolů $\boldsymbol{x}$ v opačném pořadí a zapiš $\boldsymbol{r}$ na výstup; vrať se na bod 3
7. **<u>Syntaktická chyba</u>**; ukonči algoritmus

6.5 LR syntaktický analyzátor

LR syntaktický analyzátor je jeden z nástrojů syntaktické analýzy, který umožňuje provádět syntaktickou analýzu zdola nahoru a simuluje tak konstrukci derivačního stromu od listů ke kořeni. Zkratka LR vznikla podle principu jeho fungování, protože čte řádky vstupního textu zleva doprava, shora dolů bez návratů (anglicky ***L**eft to right*) a jeho výstupem je nejpravější derivace v převráceném pořadí (anglicky ***R**ightmost derivation in reverse*). Odtud tedy zkratka LR syntaktický analyzátor.

Při syntaktické analýze je nutné se typicky vypořádat se dvěma základními problémy, které již byly zmíněny, a to pravidla se stejnou pravou stranou, u kterých je problém deterministicky určit, které pravidlo se má aplikovat, a nejednoznačné gramatiky, keré umožňují zkonstruovat více než jeden derivační strom pro stejný řetězec. Tyto problémy byly řešeny u analýzy shora dolů typicky převodem gramatiky na LL gramatiku, a poté již bylo možné použít rekurzivní sestup nebo prediktivní syntaktickou analýzu. LR syntaktický analyzátor používá ke své činnosti rozšířený zásobníkový automat a tabulku, která se skládá ze dvou částí, akční části a přechodová části. Na výstupu pak vytváří pravý rozbor vstupního řetězce, což je reverzovaná posloupnost pravidel, která je použita v nejpravější derivaci. Samotnou analýzu obsluhuje podobně jako u prediktivní syntaktické analýzy jeden algoritmus, ale konstrukce rozšířeného zásobníku a především pak celé tabulky je velmi složitá, a proto je LR syntaktický analyzátor náročný na implementaci. Na druhou stranu má větší výpočetní sílu, než analyzátory shora dolů založené na LL gramatikách nebo jeho mnohem slabší alternativa pro analýzu zdola nahoru, precedenční syntaktický analyzátor.

6.6 Precedenční syntaktický analyzátor

Precedenční syntaktický analyzátor je nástroj pro syntaktickou analýzu zdola nahoru, který je řízen velmi jednoduchým algoritmem a rozhoduje se na základě tabulky, která definuje především vzájemnou precedenci a asociativitu operátorů používaných v gramatice. Odtud vznikl název precedenční tabulka. Výpočetní síla tohoto analyzátoru je nesrovnatelně malá s LR syntaktickým analyzátorem, ale na druhou stranu je velmi oblíbený, protože je snadný na implementaci a na rozdíl od analyzátorů shora dolů se dokáže velmi dobře vypořádat se zpracováním matematických výrazů. Gramatiku pro základní matematické operace lze popsat snadno několika pravidly, ale při analýze shora dolů dochází k problému nejednoznačnosti a více derivačním stromům, z nichž správným by byl pouze ten, který dodržuje precedenci operátorů, a právě s tímto se precedenční analyzátor snadno vypořádá. To je jeden z důvodů, proč práce v praktické části spojuje univerzální prediktivní syntaktický analyzátor shora dolů, který je složitý na konstrukci, ale ne tolik jako LR analyzátor, pro analýzu syntaktické struktury kódu spolu s precedenčním analyzátorem zdola nahoru pro zpracování výrazů. Tento přístup naplňuje jednu z myšlenek gramatických systémů, kdy více přispěvatelů s částečným řešením dokáže společně vyřešit celý problém, aniž by se muselo konstruovat jedno souhrnné složité řešení, což by byl v tomto případě LR analyzátor.

Jedním z důvodů, proč je síla precedenčního analyzátoru tak malá, jsou zavedená omezení, která požadují, aby v gramatice neexistovalo více pravidel se stejnou pravou stranou a zároveň gramatika neobsahovala epsilon pravidla.

Algoritmus precedenčního analyzátoru požaduje na vstupu precedenční tabulku pro gramatiku $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ a vstupní řetězec tokenů $\boldsymbol{x} \in \boldsymbol{T}^*$. Výstupem je potom pravý rozbor pro $\boldsymbol{x}$, pokud je $\boldsymbol{x}$ řetězcem jazyka $\boldsymbol{L}(\boldsymbol{G})$, $\boldsymbol{x} \in \boldsymbol{L}(\boldsymbol{G})$, jinak syntaktický chyba.

1. Ulož symbol $ na zásobník
2. Nechť $\boldsymbol{a}$ je terminál nejblíže vrcholu zásobníku a $\boldsymbol{b}$ je aktální token na vstupu
3. Pokud $\boldsymbol{a} = \$$ a zároveň $\boldsymbol{b} = \$$, potom **úspěch**; ukonči algoritmus
4. Pokud $\boldsymbol{Tabulka}[\boldsymbol{a}, \boldsymbol{b}] = ' = '$, potom ulož $\boldsymbol{b}$ na zásobník; načti další symbol $\boldsymbol{b}$ ze vstupu; vrať se na bod 2
5. Pokud $\boldsymbol{Tabulka}[\boldsymbol{a}, \boldsymbol{b}] = ' < '$, potom zaměň $\boldsymbol{a}$ za $'\boldsymbol{a} < '$ na vrcholu zásobníku; ulož $\boldsymbol{b}$ na zásobník; načti další symbol $\boldsymbol{b}$ ze vstupu; vrať se na bod 2
6. Pokud $\boldsymbol{Tabulka}[\boldsymbol{a}, \boldsymbol{b}] = ' > '$ a zároveň $' < \boldsymbol{y}'$ se nachází na vrcholu zásobníku a zároveň existuje pravidlo $\boldsymbol{r}: \boldsymbol{A} \rightarrow \boldsymbol{y} \in \boldsymbol{P}$, potom zaměň $' < \boldsymbol{y}'$ za $\boldsymbol{A}$ na zásobníku; vypiš pravilo $\boldsymbol{r}$ na výstup; vrať se na bod 2
7. **Syntaktická chyba**; ukonči algoritmus

Samotná precedenční tabulka je tvořena maticí průniku všech terminálů. Řádky tabulky reprezentují terminály na vrcholu zásobníku ($\boldsymbol{a_i}..\boldsymbol{a_n}$ pro $\boldsymbol{i} = \boldsymbol{1}..\boldsymbol{n}$) a záhlaví sloupců představuje terminály na vstupu ($\boldsymbol{a_j}..\boldsymbol{a_n}$ pro $\boldsymbol{j} = \boldsymbol{1}..\boldsymbol{n}$). Jejich vzájemný průnik (jednotlivé buňky) může nabývat hodnot $\{<, =, >, \boldsymbol{nic}\}$. Místo teoretických ukázek bude popsána skutečná precedenční tabulka na gramatice, která byla využita při implementaci praktické části práce. Tato gramatika/tabulka umožňuje zpracovat matematické výrazy obsahující základní operace $(+, -, *, /)$ a všechny porovnávací operátory $(<, >, <=, >=, ==, !=)$. Dále pak dovoluje zkontrolovat libovolné zanoření kulatých závorek a symbol $\boldsymbol{i}$ reprezentuje identifikátor, kterým může být číselná hodnota nebo proměnná.

Nechť $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{E})$ je gramatika pro precedenční syntaktický analyzátor použitý v praktické části práce, kde:

$\boldsymbol{N} = \{\boldsymbol{E}\}$,

$\boldsymbol{T} = \{+, -, *, /, <, >, <=, >=, != , (,), \boldsymbol{i}, \$\}$,

$\boldsymbol{P} = \{$
$\mathbf{1}: \boldsymbol{E} -> \boldsymbol{E} + \boldsymbol{E}$, $\mathbf{7}: \boldsymbol{E} -> \boldsymbol{E} <= \boldsymbol{E}$,
$\mathbf{2}: \boldsymbol{E} -> \boldsymbol{E} - \boldsymbol{E}$, $\mathbf{8}: \boldsymbol{E} -> \boldsymbol{E} >= \boldsymbol{E}$,
$\mathbf{3}: \boldsymbol{E} -> \boldsymbol{E} * \boldsymbol{E}$, $\mathbf{9}: \boldsymbol{E} -> \boldsymbol{E} == \boldsymbol{E}$,
$\mathbf{4}: \boldsymbol{E} -> \boldsymbol{E} / \boldsymbol{E}$, $\mathbf{10}: \boldsymbol{E} -> \boldsymbol{E} != \boldsymbol{E}$,
$\mathbf{5}: \boldsymbol{E} -> \boldsymbol{E} < \boldsymbol{E}$, $\mathbf{11}: \boldsymbol{E} -> (\boldsymbol{E})$,
$\mathbf{6}: \boldsymbol{E} -> \boldsymbol{E} > \boldsymbol{E}$, $\mathbf{12}: \boldsymbol{E} -> \boldsymbol{i}\}$

Symboly na vstupu

Symboly na zásobníku

	+	−	∗	/	<	>	<=	=>	==	!=	(	)	*i*	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
−	>	>	<	<	>	>	>	>	>	>	<	>	<	>
∗	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
=>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<	<	<	<	<	>	>	<	>	<	>
!=	<	<	<	<	<	<	<	<	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	

Tabulka 4: Tabulka symbolů precedenční analýzy gramatiky G

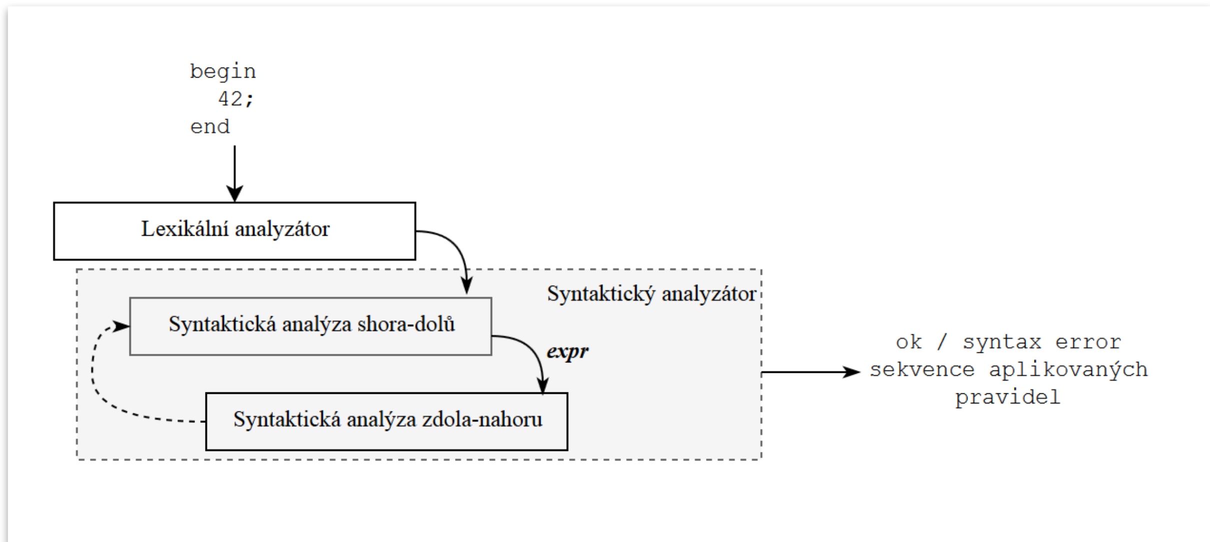
Precedenční tabulka zachycuje vztahy jednotlivých operátorů (a dalších symbolů) mezi sebou, aby je mohl analyzátor správně vyhodnotit a zpracovat v pořadí určeném jejich prioritou. Průnik operátorů se stejnou prioritou se řídí dle jejich asociativity. Levě asociativní operátory (v základní matematice to jsou všechny) mají hodnotu $' > '$, naopak pravě asociativní by měly opačnou šipku, $' < '$. Ostatní šipky porovnávají prioritu symbolů mezi sebou, přičemž se vždy porovnává priorita symbolu na

zásobníku vůči prioritě symbolu na vstupu, takže např. $'+<*'$ a naopak $'/> -'$. Prázdná buňka určuje kombinaci, kdy se symboly nemohou potkat vedle sebe, jako například dvě kulaté závorky $')\,('$ nebo identifikátor a závorka $'\boldsymbol{i}\,('$. Jediný případ, kde se vyskytuje symbol rovnítka ($'='$), je při průniku dvou párových závorek $'()'$, které jsou následně při analýze zredukovány a hlídá se tak jejich správný počet.

7 Aplikace

Praktická část práce se zaměřuje na aplikování znalostí nabytých v teoretické části práce takovým způsobem, aby byly tyto vědomosti nejen přeneseny do praxe, ale zároveň napomohly uživateli k lepšímu pochopení celé problematiky gramatických systému, překladačů a syntaktické analýzy. Vychází se z běžného problému syntaktické analýzy programovacích jazyků, které lze popsat složitou bezkontextovou gramatikou pro LR syntaktický analyzátor. Tento přístup je vhodný, pokud se celé gramatice věnuje velká pozornost a zároveň jsou k dispozici komerční nástroje, které dokáží výslednou komplexní gramatiku transformovat do funkčního analyzátoru, a udělají tak práci za nás. Pro profesionální použití je to velmi vhodná cesta, ale vyžaduje více zkušeností a znalostí. Druhý přístup nabízí použití více gramatik. Pomocí relativně jednoduché LL gramatiky lze popsat syntaktickou strukturu daného jazyka a následně s použitím LL tabulky vytvořit buďto jednoúčelový analyzátor pomocí rekurzivního sestupu nebo prediktivní syntaktický analyzátor, který pracuje nad LL tabulkou. Tato varianta je relativně snadná a dobře uchopitelná jak na pochopení, tak i na implementaci. Zůstává zde však problém s vyhodnocením matematických výrazů (často označovány jako *expressions*), protože syntaktická analýza shora dolů se špatně vypořádává s výrazy, ve kterých je nutné zohledňovat správné pořadí vyhodnocení operátorů na základě jejich priority (precedence) a dalšími záležitostmi, jako je např. libovolný počet kulatých závorek ve výrazech atd. Pro jejich zpracování je právě vhodnější přístup zdola nahoru, kterým pracuje LR syntaktický analyzátor, ale jak již bylo řečeno, vytvářet pouze kvůli výrazům jednu komplexní gramatiku, navíc s nutností implementovat složitý LR analyzátor, je pro nekomerční (příp. didaktické) účely zbytečné. Nabízí se tedy možnost nechat LL gramatiku pro popis syntaktické struktury programu a pro vyhodnocení výrazů navrhnout druhou gramatiku. Zde přichází opět v úvahu využít LR analyzátor, jehož gramatika by byla jednoduchá, protože by popisovala pouze výrazy, ale opět by byla nutná implementace celého analyzátoru, což by situaci vrátilo zpět na začátek k tomu, že je možné celý jazyk popsat gramatikou pro LR analyzátor. Druhá varianta je použít nejednoznačnou bezkontextovou gramatiku popisující pouze matematické výrazy, práci s operátory, čísly a příp. identifikátory, jejíž nejednoznačnost by za normálních okolností vadila, ale protože je použit precedenční syntaktický analyzátor, který se s tímto problémem dokáže vyrovnat, je vše v pořádku. Výsledná gramatika tak bude opravdu jednoduchá, velmi čitelná a přitom bez problému dokáže popsat požadovanou strukturu výrazů. Precedenční analyzátor je velmi jednoduchý nástroj pro syntaktickou analýzu zdola nahoru, který by někdo mohl označit za triviální a nepoužitelný. Pro složité gramatiky nebo pro celou syntaktickou analýzu by to byla pravda, ale jako jednoúčelový nástroj pro zpracování, v tomto případě matematických výrazů, je velmi vhodný. Zároveň je snadný na implementaci a z didaktických účelů donutí uživatele více přemýšlet nad vztahy mezi operátory, jejich prioritou, možnostech výskytu atd.

Výsledné dva protichůdné přístupy se vzájemně výborně doplňují. Samy o sobě jsou jednoduché a dohromady umožňují provést syntaktickou analýzu vstupního kódu požadovaného jazyka.



Obrázek 10: Schéma syntaktické analýzy aplikace

Jediným problémem tak zůstává, jak tyto dvě metody spojit, aby z nich vzniklo jedno fungující řešení. V praxi se zpravidla aplikuje přístup syntaxí řízeného překladu, kdy je syntaktický analyzátor hlavním jádrem překladače a řídí činnost ostatních komponent, jako je lexikální a sémantický analyzátor, generátor vnitřního kódu a další. Při zkoumání podobných přístupů v praxi by se pravděpodobně ukázalo, že hlavním řídícím prvkem je analýza shora dolů, která si nějakým způsobem volá metody precedenčního analyzátoru, a vše je implementované pouze za účelem funkčnosti, ale nikdo už nepřemýšlel nad tím, jestli lze toto propojení formálně specifikovat a blíže zkoumat. I zde je aplikován stejný model, protože dává smysl, aby analýzu řídil syntaktický analyzátor shora dolů. Hlídá a kontroluje strukturu celého kódu, takže ví, kde se mohou vyskytnout výrazy. Protože práce nezkoumá celý problém pouze po praktické stránce, ale především i po té teoretické, je zaveden v těchto místech do gramatiky pomocný pseudoterminál, nazvaný $\boldsymbol{expr}$ (od anglického *expression*). Lze si povšimnout, že spojení těchto dvou gramatik pomocným pseudoterminálem vede přímo ke gramatickým systémům. Možností je jistě více, ale nabízí se zvolení PC gramatického systému právě kvůli pomocnému pseudoterminálu $\boldsymbol{expr}$, který nemá přímý význam ve struktuře jazyka, ale může posloužit právě jako komunikační symbol mezi dvěma komponentami PC gramatického systému. Tento pseudoterminál se může vyskytnout pouze v první komponentě, kterou je LL gramatika pro syntaktickou analýzu shora dolů. V gramatice precedenční analýzy se vyskytnout nemůže, protože by nedávalo příliš smysl, aby se při analýze výrazu znovu volala analýza výrazu. Tato analýza buďto skončí nebo se jedná o jeden složitější výraz, ale vždy se zpracuje najednou. Je tedy patrné, že celé řešení nejlépe pasuje na centralizovaný PC gramatický systém s návraty. Pouze první řídící komponenta generuje komunikační symboly, pomocí kterých si vyžádá práci druhé komponenty, tedy precedenční analýzy, která se po vrácení výsledku resetuje do výchozího stavu (neukládá si rozpracované řešení) a opět čeká, až bude znovu zavolána.

Gramatika pro precedenční analyzátor byla podrobně představena již v dřívější kapitole 6.6 o precedenční analýze, takže zde již nebude znovu uvedena. V aplikaci byla použita přesně tato konkrétní gramatika bez dalších úprav. Naopak řídící část syntaktické analýzy shora dolů je velmi

zajímavá, protože díky použití prediktivního syntaktického analyzátoru, který požaduje na vstupu již sestavenou LL tabulku, není tento analyzátor vázán na konkrétní gramatiku.

V rámci této praktické části byla vytvořena webová aplikace, která umožňuje zadat libovolný vstupní text, nad nímž provede syntaktickou analýzu a zobrazí uživateli posloupnost aplikovaných pravidel gramatiky. Pokud je zdrojový kód v pořádku, oznámí, že je vše *ok*. V opačném případě se výpis aplikovaných pravidel zastaví na výskytu první chyby a tato chyba je interpretována uživateli. Většinou se jedná o informaci, na kterém řádku a jakém tokenu došlo k chybě. Taková aplikace by sama o sobě byla velmi vhodnou didaktickou pomůckou, protože nevyžaduje žádnou instalaci nebo složité spouštění přes konzoli apod., jak tomu často bývá u aplikací v odborných pracích. Zároveň se uživatel dozví, jak byla pravidla postupně aplikována, a to pro obě gramatiky. Sekvence pravidel obsahuje čísla pravidel LL gramatiky při kontrole shora dolů, ale i výpis jednotlivých pravidel precedenční analýzy při přepnutí komponenty pro zpracování výrazů. Aplikace však zachází ještě dál a díky vhodné implementaci umožňuje vložit libovolnou LL gramatiku vč. epsilon pravidel. V jednotlivých kapitolách byla velmi podrobně vysvětlena konstrukce LL tabulky spolu s jednotlivými pomocnými množinami. Všechny tyto kroky jsou přeneseny do aplikace a umožňují dynamicky vygenerovat celou LL tabulku ze zadané gramatiky. Tato tabulka je potom předána syntaktickému analyzátoru při spuštění analýzy. Protože vytvoření LL tabulky a následná syntaktická analýza probíhá automaticky ze zadané LL gramatiky, je nutné klasickou definici bezkontextové gramatiky rozšířit o další atributy. Definice vstupní LL gramatiky $\boldsymbol{G_{LL}}$ je následující:

$$\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S}, \boldsymbol{Expr}, \boldsymbol{\varepsilon}, \boldsymbol{Int}, \boldsymbol{Float}, \boldsymbol{String}, \boldsymbol{Id})$$

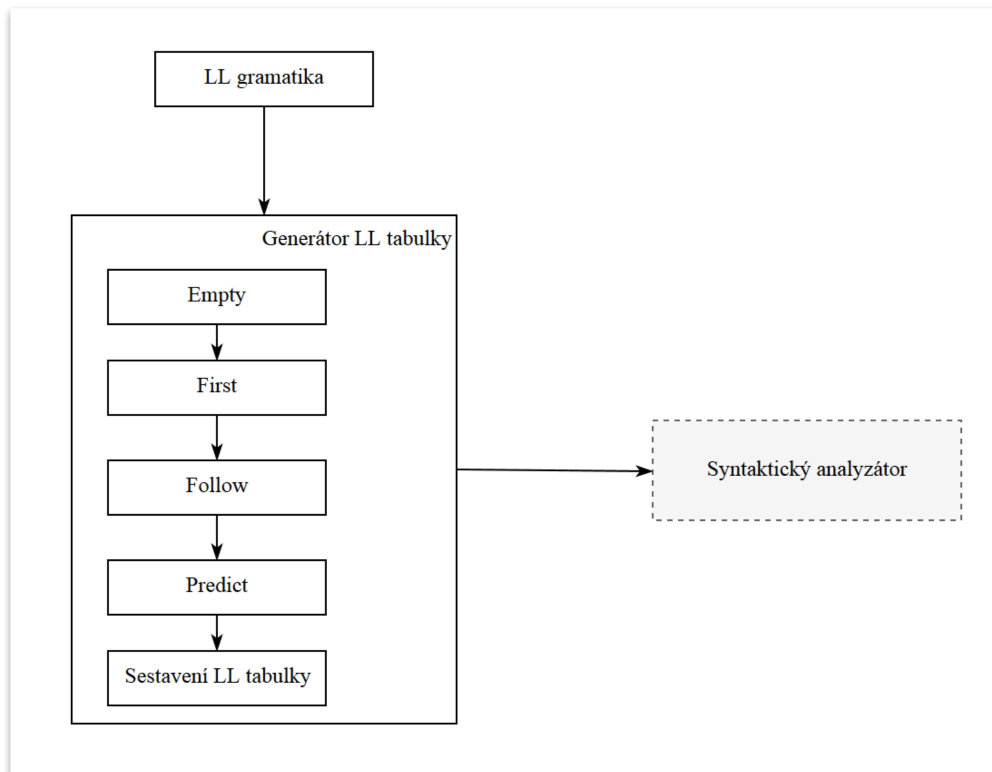
- $\boldsymbol{N}$ je konečná neprázdná množina neterminálních symbolů
- $\boldsymbol{T}$ je konečná neprázdná množina terminálních symbolů a platí, že $\boldsymbol{N} \cap \boldsymbol{T} = \emptyset$
- $\boldsymbol{P}$ je konečná množina pravidel tvaru $\boldsymbol{A} \to \boldsymbol{x}$, kde $\boldsymbol{A} \in \boldsymbol{N}, \boldsymbol{x} \in (\boldsymbol{N} \cup \boldsymbol{T} \cup \boldsymbol{Expr})^+$ (epsilon pravidlo se vytvoří výběrem zástupného epsilon terminálu $\boldsymbol{\varepsilon}$)
- $\boldsymbol{S} \in \boldsymbol{N}$ je počáteční (startující) neterminál

Následující atributy jsou volitelné a rozšiřují definici gramatiky $\boldsymbol{G_{LL}}$:

- $\boldsymbol{Expr}$ je pseudoterminál, který slouží pro přepnutí řízení syntaktické analýzy z analýzy shora dolů na analýzu zdola nahoru
- $\boldsymbol{\varepsilon} \in \boldsymbol{T}$ je vybraný terminál reprezentující epsilon pro vytvoření epsilon pravidla
- $\boldsymbol{Int} \in \boldsymbol{T}$ je vybraný terminál reprezentující celočíselný datový typ integer
- $\boldsymbol{Float} \in \boldsymbol{T}$ je vybraný terminál reprezentující číselný datový typ s plovoucí řádovou čárkou float
- $\boldsymbol{String} \in \boldsymbol{T}$ je vybraný terminál reprezentující datový typ řetězec
- $\boldsymbol{Id} \in \boldsymbol{T}$ je vybraný terminál reprezentující identifikátor

Rozšíření klasické bezkontextové gramatiky tímto způsobem bylo nezbytné, protože aplikace nabízí velkou flexibilitu, ale pro praktické použití je nutné specifikovat základní datové typy a další elementy jazyka, aby bylo možné provádět syntaktickou analýzu nad běžnými programovacími jazyky a dalšími podobnými konstrukcemi.

Následující schéma zachycuje jednotlivé bloky konstrukce LL tabulky tak, jak je LL gramatika postupně analyzována a výsledná LL tabulka je předána syntaktickému analyzátoru pro řízení analýzy shora dolů.



Obrázek 11: Schéma konstrukce LL tabulky

Obě gramatiky byly formálně zavedeny, přičemž LL gramatiku pro syntaktickou analýzu shora dolů lze označit $\boldsymbol{G_{LL}}$ a gramatiku pro precedenční analýzu zdola nahoru z předchozí kapitoly 6.6 $\boldsymbol{G_{PR}}$. Pro již zmíněný centralizovaný PC gramatický systém s návraty bude jeho formální definice následující:

$$\boldsymbol{\Gamma} = (\boldsymbol{N}, \boldsymbol{K}, \boldsymbol{T}, (\boldsymbol{S_{LL}}, \boldsymbol{P_{LL}}), (\boldsymbol{E_{PR}}, \boldsymbol{P_{PR}})), \text{ kde:}$$

- $\boldsymbol{N} = \boldsymbol{N_{LL}} \cup \boldsymbol{N_{PR}}$
- $\boldsymbol{K} = \{\boldsymbol{Expr_{LL}}\}$
- $\boldsymbol{T} = \boldsymbol{T_{LL}} \cup \boldsymbol{T_{PR}}$
- pro všechny $\boldsymbol{A} \rightarrow \boldsymbol{x} \in \boldsymbol{P_{PR}}$ platí, že **abeceda(x)** $\cap \boldsymbol{K} = \emptyset$

Dynamická LL gramatika zvyšuje přidanou hodnotu aplikace, protože uživatel má možnost experimentovat. Může si vložit vlastní gramatiku a na základě výstupů ji libovolně modifikovat a upravovat v reálném čase. Vždy před další syntaktickou analýzou se znovu vygeneruje nová tabulka, aby zůstávala stále aktuální k zadané gramatice. Aplikace dále zobrazuje uživateli sestavenou LL tabulku, takže se díky aplikaci dozví nejen, jakým způsobem byl jeho vstupní kód na základě vlastní gramatiky analyzován, ale i jak vypadá příslušná LL tabulka. Podobný princip by bylo možné aplikovat i u precedenčního analyzátoru, ale pro lepší použití aplikace bylo cílem, aby měla alespoň nějaký pevný bod, a vyhodnocování matematických výrazů je vhodnou volbou, protože se pro běžné programovací jazyky neliší. Stačí si tedy načíst webovou stránku, kde je aplikace umístěná, vložit vlastní LL gramatiku spolu se vstupním kódem a již je možné sledovat vygenerovanou LL tabulku a výsledky analýzy.

7.1 Uživatelské rozhraní

Hlavním záměrem celé aplikace je jednoduchost, intuitivní použití a co největší přínos pro uživatele. Tyto aspekty se odráží již na první pohled v samotném designu, který vychází z čisté bílé barvy, na které dobře vyniknou důležité prvky. Po načtení se zobrazí jedna stránka (tento přístup je nazýván *single page* aplikace), která je vymezena záhlavím s názvem aplikace „Dynamic Parser" a v pravém horním rohu se nachází informační tlačítko, které po stisknutí zobrazí dialogové okno s obecnými informacemi o aplikaci a precedenční tabulkou. Celá aplikace je v angličtině s výjimkou těchto obecných informací. Zápatí stránky je pak ohraničeno šedým pruhem obsahujícím kontakt a autora stránky.

Tělo stránky obsahuje vše ostatní. V pravém horním rohu se nachází tlačítka umožňující načíst vzorové příklady gramatiky a vstupního kódu. Pod mini je stránka rozdělena do dvou sloupců v poměru 2/3 a 1/3. Levá větší část obsahuje komponenty pro vložení gramatiky. Uživatel může definovat název gramatiky a následně vložit první pravidlo, které má samostatně vstup pro levou stranu a pravou stranu. Pod tímto řádkem se nachází tlačítko pro přidání dalších řádků. Jednotlivé řádky pravidel lze poté mazat tlačítkem **x**. Vstup pro vkládání pravidel není obyčejný textový HTML input, ale speciální komponenta Tags-input[3], která umožňuje vkládat text formou vizuálně oddělených tagů (štítků), které v tomto případě přímo korespondují s jednotlivými terminály a neterminály. Pokud je vložen symbol do levé strany pravidla, lze vložit pouze jeden, automaticky se obarví modře a pokládá se za neterminál. Naopak v pravé straně pravidla je po vložení tagu uživatel vyzván k výběru, jestli se jedná o terminál, neterminál nebo expression (pseudoterminál pro komunikaci mezi gramatikami) reprezentující matematický výraz. Zároveň tato komponenta podle aktuálně napsaného textu zobrazuje nabídku již existujích tagů, takže v okamžiku, kdy je již napsaná část gramatiky, je její další doplňování o to rychlejší, protože opakující se tagy není nutné znovu psát, ale po zadání prvních písmen jsou automaticky nabídnuty ke vložení. Pravý sloupec obsahuje více předdefinovaných Tags-input komponent, které zobrazují výčty jednotlivých množin. Nachází se zde množina terminálů, neterminálů, expression pseudoterminál, aliasy pro epsilon, celá čísla, reálná čísla, identifikátory a další, tedy vše, co je nutné pro definování celé gramatiky, přičemž není potřeba vyplňovat všechna pole, ale záleží na komplexnosti pravidel. Jsou-li použity pouze např. proměnné, mohou zůstat vstupy definující terminály pro celá čísla, reálná čísla a řetězce prázdné. Tyto dva sloupce jsou úmyslně vedle sebe, protože na větších monitorech se tak zobrazí celá gramatika a vedle ní její konfiguraci. Pokud se obrazovka zmenší nebo se aplikace zobrazí např. na telefonu, přeskládají se sloupce díky responzitivitě pod sebe a opět bude manipulace s aplikací pohodlnější.

Pod tímto blokem pro gramatiku se nachází sekce s LL tabulkou, kde je ve výchozím stavu umístěno pouze tlačítko pro generování samotné LL tabulky, ta je poté na tomto místě zobrazena. Dále navazuje poslední sekce, opět rozdělena na dva sloupce, ale tentokrát půl na půl. V levém sloupci je připraveno pole pro vstupní kód, který bude analyzován, spolu s tlačítkem, které analýzu vyvolá. Není nutné předem ručně generovat LL tabulku, při stisknutí tlačítka pro analýzu se vždy vygeneruje znovu. V pravém sloupci se po provedení analýzy zobrazí vstupní kód po jednotlivých očíslovaných řádcích a rozdělený na tokeny. Komentáře jsou obarveny zeleně a při analýze se přeskakují. Poté následuje výsledek analýzy (symbol ✓ pro úspěch nebo chybová zpráva) a posloupnost aplikovaných pravidel. Oba sloupce se opět při menším rozlišení obrazovky posunou pod sebe, aby nezabíraly tolik místa. Na velkém monitoru je pak vše přehledně vedle sebe, vstup i odpovídající výstup. Konkrétní ukázku aplikace lze nalézt na konci práce v sekci příloh.

---

[3] Komponenta pro automatické vytváření štítků. http://mbenford.github.io/ngTagsInput/

7.2 Implementace

Aplikace je vytvořena jako jednoduchá internetová stránka, tzv. single page aplikace (vše se nachází a funguje v rámci této jedné stránky), a implementuje celou řadu moderních webových technologií a přístupů. Samotná kostra je napsaná v HTML5[4] a pro styly jsou použity jak CSS3 soubory, tak i modernější LESS[5] soubory spolu s frameworkem Bootstrap[6]. Aby se aplikace dobře spravovala a vyvíjela, byla vytvořena jako projekt pro MS Visual Studio[7] 2013 (a novější). V tomto vývojovém prostředí se nejen dobře orientuje v adresářové struktuře projektu, ale umožňuje konfiguraci dalších funkcionalit, jako je např. automatický publish profil pro snadnější nasazení aplikace.

Hlavní a jedinou stránkou aplikace je soubor index.html. Celá aplikační logika je implementovaná pouze pomocí JavaScriptu, takže lze říct, že se jedná o čistě klientskou aplikaci. Poté, co se stránka načte v prohlížeči, nevyžaduje internetové připojení a nikam se nepřipojuje, vše funguje lokálně v prostředí prohlížeče. Nejedná se však o čistý JavaScript, ale o moderní framework AngularJS[8], který je složitější na pochopení, ale nabízí mnoho výhod, jako je například obousměrné vázání proměnných do webové stránky (tzv. two-way binding). Tyto výhody mají obecně vliv na výkon JavaScriptu, ale jeho degradace by byla citelná např. u informačního systému. U této aplikace, kde se neklade důraz na výkon a rychlost zpracování, to nijak omezuje, a lze tak využít všech výhod, které nabízí. Kolem AngularJS vznikla vývojářská komunita, takže existuje mnoho volně dostupných aplikací třetích stran, mezi které patří např. Tags-input použitý pro uživatelsky velmi přívětivé definování gramatiky pomocí štítků. Aby bylo možné jednotlivé moduly pro AngularJS lépe spravovat, využívá projekt balíčkovací systém NPM[9] a pro automatický build JavaScriptu a stylů konzolový program gulp[10]. Při vývoji aplikace potom stačí spustit v konzoli příkaz gulp, který při jakékoliv změně JavaScriptu nebo stylů znovu provede build aplikace, takže je celý klient během vývoje neustále aktuální. Před nasazením na produkci potom stačí spustit gulp s parametrem „–p“ a JavaScript se minifikuje do jednoho souboru app.js. Všechny soubory se styly (CSS i LESS) se zpracují podobným způsobem a uloží v souboru app.css. Styly využívají framework Bootstrap, jak již bylo zmíněno, a ikony jsou převzaty z Font Awesome[11].

Struktura aplikace obsahuje na nejvyšší úrovni soubor index.html a dva adresáře WebClient a Views. Ostatní soubory mají význam pouze pro Visual Studio a není nutné je zde rozebírat. Složka Views obsahuje html soubory, které představují dialogová okna zobrazující se v aplikaci. WebClient reprezentuje celého JavaScript klienta včetně všech stylů, modulů atd. Uvnitř této složky se nachází soubor package.json a dvě související složky node_modules a webpack, které obsahují moduly a konfigurace pro AngularJS a NPM. Dále jsou zde dva adresáře. První složka se jmenuje build a obsahuje výsledné minifikované soubory a fonty. A konečně složka src, ve které se nachází jednotlivé JavaScript soubory a styly. Základním souborem je main.js, který spojuje soubory dohromady. Samotná logika aplikace je potom rozdělena do kontrolérů umístěných ve složce controllers a pomocné funkce a konstanty jsou ve složce helpers. Veškeré styly použité v aplikaci se nachází ve složce style. Jsou zde uloženy soubory pro komponenty Bootstrapu, dialogová okna z modulu

---

[4] Nový standard HTML. https://www.w3.org/TR/html5/
[5] CSS soubory rozšířené o dynamické chování jako jsou proměnné, operace, funkce http://lesscss.org/
[6] HTML, CSS, and JS framework pro responzivní design. http://getbootstrap.com/
[7] Integrované vývojové prostředí firmy Microsoft. https://www.visualstudio.com/cs/
[8] Moderní framework pro JavaScript. https://angularjs.org/
[9] Balíčkovací systém pro JavaScript. https://www.npmjs.com/
[10] Nástroj umožňující automatizaci různých úkonů pro vývoj (např. build) https://www.npmjs.com/package/gulp
[11] Balík vektorových ikon. http://fontawesome.io/

ngDialog, Font Awesome a vlastní třídy pro samotnou aplikaci. Celý systém stylů a knihoven je udělán tímto způsobem, aby nebylo nutné nic stahovat z externích zdrojů z internetu, ale vše bylo při vývoji k dispozici offline a u výsledné aplikace vystavené na internet stačilo, aby si načetla své vlastní lokální zdroje.

Implementace výkonné logiky aplikace je uložena v jednotlivých kontrolérech. Na nejvyšší úrovni se nachází system-controller.js, jehož metody a proměnné jsou přístupné ostatním kontrolérům. Jsou zde umístěny metody pro základní práci s jednotlivými množinami, načítání terminálů, neterminálů atd. Zbylá funkcionalita je rozdělená na tři části. Celá lexikální analýza se nachází v souboru lexical-analysis.controlles.js. Sekvence metod pro vytvoření LL tabulky je v LLtable-controller.js a závěrečná syntaktická analýza je implementována v syntax-analysis-controller.js. Není nutné rozebírat implementaci a fungování jednotlivých metod, protože přesně kopírují funkcionalitu odpovídajících algoritmů, které byly podrobně probrány v jednotlivých kapitolách, a pouze ji upravují pro JavaScript. Za zmínku stojí, že lexikální analýza se neprovádí, jak tomu často bývá, průběžně, kdy syntaktická analýza řídí překlad a dotazuje se dle potřeby vždy na další token, ale provede se na začátku pro celý vstupní kód. Je to z toho důvodu, aby byl vstupní kód již ve formě tokenů celý k dispozici při dalších analýzách a bylo jej možné zobrazit v této podobě na výstupu. Po lexikální analýze se provede sestavení LL tabulky a poté již proběhne najednou celá syntaktická analýza.

7.3 Nasazení a vývoj

Jedná o čistě klientskou aplikaci s výchozím souborem index.html, takže pro nasazení stačí překopírovat obsah složky WebSite na server a vše bude fungovat automaticky samo. Toto platí pro případ, že je k dispozici publikovaná verze aplikace. V projektu pro MS Visual Studio je pro ulehčení práce vytvořený odpovídající publish profil.

Zprovoznění aplikace pro vývoj na lokálním stroji není úplně jednoduché vzhledem k velkému množství použitých technologií, doplňků atd. Návod na instalaci celého vývojového prostředí je uveden v jednotlivých na sebe navazujících bodech, které by mělo stačit následovat. Vývojové prostředí se skládá ze dvou částí, první z nich je MS Visual Studio 2013/15, které ale není nezbytně nutné. Celý projekt je vytvořený pro toto prostředí, ale jednotlivé soubory lze editovat v libovolném textovém editoru. Důležitější a složitější je instalace webového klienta (návod je pro MS Windows).

1. Nainstalovat Node.js (https://nodejs.org/dist/v0.12.7/)
2. Přidat do systémových proměnných proměnnou GYP_MSVS_VERSION s hodnotou „2013“ (verze MS Visual Studia, některé použité balíčky tuto hodnotu potřebují)
3. Nainstalovat Python (verzi 2.7.10)
4. Přidat Python do Path (např. „C:\Python27\;C:\Python27\Scripts;“)
5. Pokud používáte Git, spustit git bash v adresáři WebClient, jinak klasicky cmd
6. Spustit příkaz: „npm install gulp –g“
7. Spustit příkaz: „npm install“

Příkazy pro vývoj:

„gulp“ Automatický build a tzv. livereload, který hlídá změny ve zdrojových souborech a automaticky provede build při každé změně.

„gulp –p“ Provede build JavaScriptu a stylů pro produkční prostředí, vytvoří tedy minifikované verze.

8 Závěr

Práce postupně obsáhla několik různých samostatných témat, které se ve výsledku spojily do jednoho funkčního celku a skvěle se vzájemně doplňují. Nejdříve se práce věnuje základu formálních jazyků, důležitým pojmům a formalismům. Tato kapitola, stejně jako ostatní, by jistě dokázala sama obsáhnout celou práci, ale bylo důležité zmínit alespoň to nejzákladnější pro lepší zasazení čtenáře do kontextu a sjednocení jeho znalostí s terminologií zbytku práce. Byly zde zmíněny tedy takové pojmy jako abeceda, řetězec, Chomského hierarchie a další. Po tomto základu mohla bez problému navázat kapitola o regulárních gramatikách. Význam této kapitoly je v kontextu celé práce trochu skrytý a někdo by mohl namítnout, že vložit tuto kapitolu do práce bylo zbytečné. Nicméně základy práce s řetězci a jazyky se velmi dobře vysvětlují nad regulárními jazyky, protože tvoří základní stavební kameny složitějších typů jazyků a úzce s nimi souvisejí konečné automaty, které jsou jedním z formálních prostředků regulárních jazyků. Zároveň byl konečný automat použit pro implementaci lexikální analýzy v praktické části práce, která není z pohledu práce významná, nicméně pro funkčnost aplikace je neopomenutelná. Následující kapitola navázala vyšší třídou jazyků Chomského hierarchie, a to bezkontextovými jazyky, resp. gramatikami, se kterými se poté pracuje ve zbytku celé práce. Bezkontextové gramatiky jsou velmi významné a dovolují popsat mimo jiné většinu programovacích jazyků, avšak vyskytuje se zde několik komplikací, z nichž jedna se týká nejednoznačnosti. K této situaci dochází, pokud existuje více pravidel gramatiky se stejnou levou stranou pravidla. I tento problém má své řešení, jako je např. zavedení determinismu nebo speciální LL gramatiky, které se využívají při syntaktické analýze shora dolů. S bezkontextovými gramatikami souvisí zásobníkové automaty, které jsou jedním z jejich formálních aparátů. Jedná se o konečné automaty rozšířené o zásobník a opět i tyto prostředky byly následně aplikovány v praktické části.

Po bezkontextových gramatikách přichází na řadu jádro práce, bezkontextové gramatické systémy. Tyto systémy vznikly v průběhu druhé poloviny 20. století a umožňují formálně popsat a definovat spolupráci více gramatik při řešení společného problému. V mnoha ohledech jsou podobné principy již dávno v praxi uplatňovány, ale nemají zavedené žádné formalismy, protože je hlavním cílem funkčnost a nedává se prostor teorii. Jenomže bez teoretického základu lze špatně provádět další výzkum a rozvoj takovýchto systémů, obzvláště pokud jde např. o validaci a verifikaci. Jedním z nejčastějších příkladů spolupráce více gramatik je syntaktická analýza. Lze provést analýzu vstupního kódu pomocí jedné komplexní gramatiky, ale to vyžaduje definici složité gramatiky a zároveň i komplexního nástroje, který ji dokáže zpracovat. V mnoha ohledech se jeví jako vhodnější přístup mít více gramatik, z nichž každá dokáže řešit část vstupního textu (jazyka) a společně zvládnou provést kompletní analýzu. Jako jeden konkrétní příklad za všechny je možné zmínit kombinaci syntaktické analýzy shora dolů pro kontrolu syntaktické stavby jazyka s analýzou zdola nahoru pro zpracování a vyhodnocení matematických výrazů, kde se klade velký důraz na správné pořadí zpracování matematických operací dle priority operátorů. Přesně pro tuto problematiku jsou gramatické systémy určeny, protože jejich základní myšlenka spočívá v tom, že úspěchu nemusí dosáhnout jeden složitý systémem, ale kombinace a spojení dílčích řešení. Vycházejí tak z velmi inspirativních modelů reálného světa, jako je například tzv. „problém tabule“ (anglicky *The Black Board Model*), ve kterém skupina lidí řeší společný problém u tabule a různě se střídají podle toho, kdo má nějaký nápad, kterým by mohl přispět, a posunout tak řešení blíže úspěchu. Vznikly dva základní typy gramatických systémů. První typ umožňuje distribuované zpracování a nazývá se kooperující distribuovaný gramatický systém, zkráceně CD gramatický systém (z anglického *Cooperating Distributed Grammar System*). Jednotlivé komponenty systému (takto se v terminologii gramatických systémů nazývají gramatiky daného systému) pracují sekvenčně na jedné sdílené větné

formě, kterou postupně upravují a dle různých pravidel se střídají při jejím zpracování. Naopak druhý typ gramatických systémů je představitelem paralelního zpracování a tento systém se nazývá paralelně komunikující gramatický systém, zkráceně PC gramatický systém (z anglického *Parallel Communicating Grammar System*). Komponenty tohoto systému pracují všechny současně, tedy paralelně, na svých vlastních větných formách. Každá komponenta tedy upravuje pouze svoji větnou formu, ale mohou mezi sebou komunikovat a předávat si mezivýsledky pomocí tzv. komunikačních symbolů. Velký potenciál gramatických systémů spočívá právě v těchto dvou přístupech, protože zavedly formální prostředky pro oblasti, které jsou v dnešní době stále více aktuální. Díky současným trendům v IT a komunikačních technologiích je paralelní a distribuované zpracování velkým tématem, a do budoucna lze očekávat jedině vzestupné tendence. Proto je výzkum v těchto oblastech, dle mého názoru, nejen důležitý, ale i nutný a gramatické systémy jsou jeden z prostředků, který by mohl přinést další možnosti, ať už v oblasti překladačů nebo i jiných odvětvích, jako je lingvistika, molekulární biologie a další.

Druhá polovina práce je věnována především praxi, a to v podobě syntaktické analýzy a syntaxí řízeném překladu. Jednotlivé kapitoly postupně popisují teorii nutnou pro implementaci syntaktického analyzátoru, porovnávají různé přístupy a možnosti. Mým cílem bylo vytvořit aplikaci, která kombinuje dva přístupy, tedy dvě různé gramatiky, syntaktické analýzy, a to syntaktickou analýzu shora dolů s analýzou zdola nahoru a uvažovat je v kontextu gramatických systémů. Pro syntaktickou analýzu shora dolů jsou velmi vhodné LL gramatiky, které sice mají menší vyjadřovací sílu, než obecné bezkontextové gramatiky, ale umožňují velmi snadno implementovat syntaktický analyzátor. Existují dva nejčastější přístupy, rekurzivní sestup a prediktivní syntaktická analýza. Oba přístupy vycházejí z tzv. LL tabulky, která na průniku všech terminálů s neterminály gramatiky definuje, jaké pravidlo z množiny pravidel gramatiky se má aplikovat pro danou kombinaci terminálu na vstupu s aktuálně zpracovávaným neterminálem na zásobníku. Ačkoli je rekurzivní sestup velmi intuitivní a jednoduchý (každý neterminál je reprezentován metodou a tyto metody se dle pravidel gramatiky mezi sebou rekurzivně volají, dokud nezkontrolují celý vstupní řetězec), rozhodl jsem se pro prediktivní syntaktický analyzátor. Tento přístup sice vyžaduje implementaci zásobníku a LL tabulky, ale samotná analýza je řízena jedním algoritmem a výsledný analyzátor není nijak závislý na konkrétní LL tabulce, což se ukázalo velmi užitečné. Pro vytvoření LL tabulky je nutné postupně sestavit několik pomocných množin, jejichž podrobnému popisu jsou věnovány jednotlivé podkapitoly.

Syntaktická analýza zdola nahoru má také své dva zástupce, z nichž komplexním řešením je LR syntaktický analyzátor, který pokud má k dispozici komplexní gramatiku, dokáže provést analýzu celého vstupu. Je tak velmi vhodný pro profesionální a komerční řešení, ale je složitý na implementaci. Zároveň pokud by již byl použit, bylo by snazší ho aplikovat pro celou analýzu a ne pouze pro část. Proto je pro účely této práce vhodnější druhý přístup, tzv. precedenční syntaktický analyzátor. Jedná se o velmi jednoduchý aparát, který nemá příliš velkou vyjadřovací sílu, ale je velmi vhodný např. pro vyhodnocování matematických výrazů, což je přesně část kódu, se kterou má problém prediktivní syntaktická analýza shora dolů. Existují tak tedy dva přístupy, z nichž každý pracuje jiným způsobem a je vhodný pro analýzu jiné části vstupního kódu, což jde přesně ruku v ruce s myšlenkou gramatických systémů. Velmi se zde nabízí konkrétní řešení, a to centralizovaný PC gramatický systém. Tento systém je speciální v tom, že pouze hlavní řídící komponenta může generovat komunikační symboly, a má tak možnost si dle potřeby vyžádat práci ostatních komponent. Řídící komponenta je v tomto případě analýza shora dolů, která kontroluje syntaktickou strukturu celého jazyka a pro zpracování matematických výrazů si zavolá analýzu zdola nahoru, přičemž pro komunikaci mezi komponentami byl zaveden zvláštní pseudoterminál.

Výsledná aplikace tak umožňuje uživateli provádět syntaktickou analýzu jeho vlastního vstupního textu (předpokládá se nějaká forma programovacího jazyka) na základě spolupráce těchto dvou gramatik. Hlavním cílem celé aplikace je jednoduchost použití a co největší přidaná hodnota pro uživatele, proto se jedná o webovou aplikaci. Uživatel nemusí nic instalovat ani složitě spouštět, pouze se připojí na webovou stránku, kde je daná aplikace umístěná. Aplikace je navržená jako jednoduchá responzivní stránka (tzv. single page aplikace) s čistým bílým designem, aby byla co nejjednodušeji použitelná a přehledná. Uživatel má dále možnost snadno pomocí tagů (štítků) definovat svoji vlastní LL gramatiku pro analýzu shora dolů. To je možné díky tomu, že je analýza shora dolů implementovaná pomocí prediktivní syntaktické analýzy a není závislá na konkrétní LL tabulce. Uživatel tedy nadefinuje svoji LL gramatiku, vloží vstupní kód, nad kterým chce provést syntaktickou analýzu, a aplikace si již sama dynamicky sestaví LL tabulku a provede analýzu. Jako výsledek zobrazí uživateli úspěch nebo chybové hlášení a posloupnost aplikovaných pravidel jak LL gramatiky, tak i precedenčního analyzátoru. Uživatel má tak k dispozici nejen nástroj pro syntaktickou analýzu, ale uvidí způsob aplikace pravidel a vygenerovanou LL tabulku. Aplikace tedy slouží především pro didaktické použití a měla by uživateli napomoct k lepšímu pochopení problematiky překladačů, bezkontextových gramatik, syntaktické analýzy a v neposlední řadě gramatických systémů. Zároveň byla aplikace implementována pomocí celé řady moderních webových technologií, jako je HTML5, CSS3, LESS, Bootstrap, JavaScript, AngularJS a dalších, které jsou podrobněji popsány v kapitole o aplikaci. V sekci příloh se potom nachází reálná ukázka výsledné aplikace a jejího použití.

Práce zasahuje do velkého množství témat a otevírá mnoho dveří dalšímu výzkumu a rozvoji. V první řadě se nabízí teoretický i praktický výzkum paralelismu a distribuce výkonu gramatických systémů při syntaktické analýze. Zjišťovat, jaký dopad by mělo použití více gramatik (komponent), ze kterých by se sestavil a popsat komplexní gramatický systém. Tento systém v ideálním případě přenést do praxe, ať už pomocí paralelních výpočtů, pokud by se jednalo o PC gramatický systém, nebo distribuovat výkon mezi větší množství výpočetních stanic u CD gramatických systémů. Dalo by se tímto způsobem dosáhnout většího výkonu? Mělo by význam zavést tyto systémy do praxe vzhledem ke stávajícím komerčním přístupům? Okolo gramatických systémů je mnoho otázek, jejichž zodpovězení by mohlo přinést řadu nových a významných objevů nebo naopak zjištění, že jsou velmi zajímavé z teoretického pohledu („na papíře“), ale reálně nepoužitelné.

Další možností je rozvoj samotné aplikace, která díky implementaci pomocí moderních technologií tak rychle nezestárne a nabízí celou řadu způsobů, jak ji dále rozvinout. Jedna varianta by mohla být rozšíření výstupu analýzy o další podrobné informace, které by tak mohly poskytnout hlubší porozumění celé problematice. Nebo se zaměřit na dynamickou stránku a umožnit definici i druhé gramatiky, která pracuje zdola nahoru, což by mohlo být velmi zajímavé. Uživatel by mohl definovat kromě LL gramatiky i precedenční gramatiku, resp. precedenční tabulku. Případně by se dala nahradit precedenční analýza silnějším nástrojem, např. LR syntaktickým analyzátorem. Poté by šla zkoumat síla LL gramatiky vůči bezkontextové gramatice pro LR analyzátor. Pokud by se implementace provedla správně, bylo by velmi pravděpodobné, že by se dalo provést spojení obou analýz, jak je tomu teď, nebo by mohla být LR gramatika tak komplexně napsána, že by převzala sílu LL gramatice a provedla pak analýzu sama. Stejně jako samotné gramatické systémy, tak i aplikace nabízí mnoho možností, jak v této práci pokračovat.

Literatura

[1] MEDUNA, Alexander a Petr ZEMEK. *Regulated grammars and automata.*, 694 stran. ISBN 1493903683.

[2] MEDUNA, Alexander. *Formal languages and computation: models and their applications.* 315 stran. ISBN 9781466513457.

[3] MEDUNA, Alexander. *Elements of compiler design.* Boca Raton, FL: Auerbach Publications, 2008, 286 stran. ISBN 1420063235.

[4] TECHET, Jiří, Tomáš MASOPUST a Alexander MEDUNA. Modern Formal Language Theory: Cooperating Distributed Grammar Systems [online]. Brno, 2007 [cit. 2017-02-01]. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:09-cdgspres.pdf

[5] TECHET, Jiří, Tomáš MASOPUST a Alexander MEDUNA. Modern Formal Language Theory: Parallel Communicating Grammar Systems [online]. Brno, 2007 [cit. 2017-02-01]. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:10-pcgspres.pdf

[6] ROZENBERG, Grzegorz. Cooperating grammar systems. *Mathematical Foundations of Computer Science 1978: Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 1978, strany. 364-373. ISBN 9783540089216.

[7] ROZENBERG, Grzegorz a Arto SALOMAA. *Handbook of formal languages: background and application.* Berlin: Springer-Verlag, 1997, 528 stran. ISBN 3540614869.

[8] NII, Penny. *The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures.* AI Magazine Volume 7 Number 2, 1986, strany 38–53

[9] WEBBER, Adam. *Formal language: a practical introduction.* Wilsonville, Or.: Franklin, Beedle & Associates, 2008, 388 stran. ISBN 978-1590281970.

[10] LINZ, Peter. *An introduction to formal languages and automata.* 5th ed. Sudbury, MA: Jones & Bartlett Learning, 2012, 437 stran. ISBN 978-1449615529.

[11] BROOKSHEAR, J. Glenn. *Theory of computation: formal languages, automata, and complexity.* Redwood City, Calif.: Benjamin/Cummings Pub. Co., 1989, 320 stran. ISBN 978-0805301434.

[12] SIPSER, Michael. *Introduction to the theory of computation.* 3rd Ed. Boston, MA: Course Technology Cengage Learning, 2012, 480 stran. ISBN 978-1133187790.

[13] CSUHAJ-VARJÚ, E. *Grammar systems: A grammatical approach to distribution and cooperation.* Psychology Press, 1994, 246 stran. ISBN 2881249574.

[14] PAUN, Gheorghe a Arto SALOMAA. *New trends in formal languages: control, cooperation, and combinatorics.* New York: Springer, 1997, 474 stran. ISBN 3540628444.

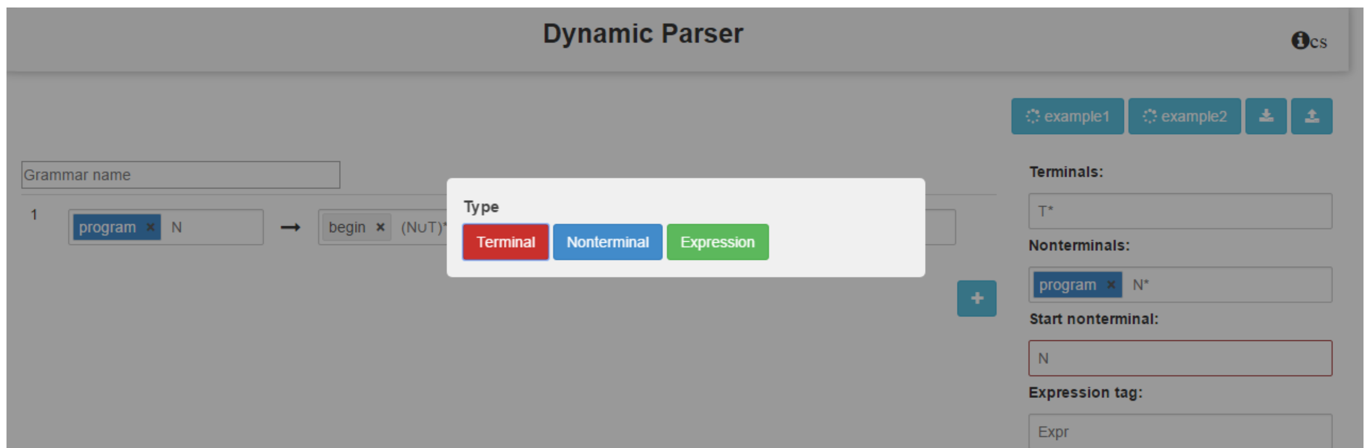
Příloha A

Obsah CD:

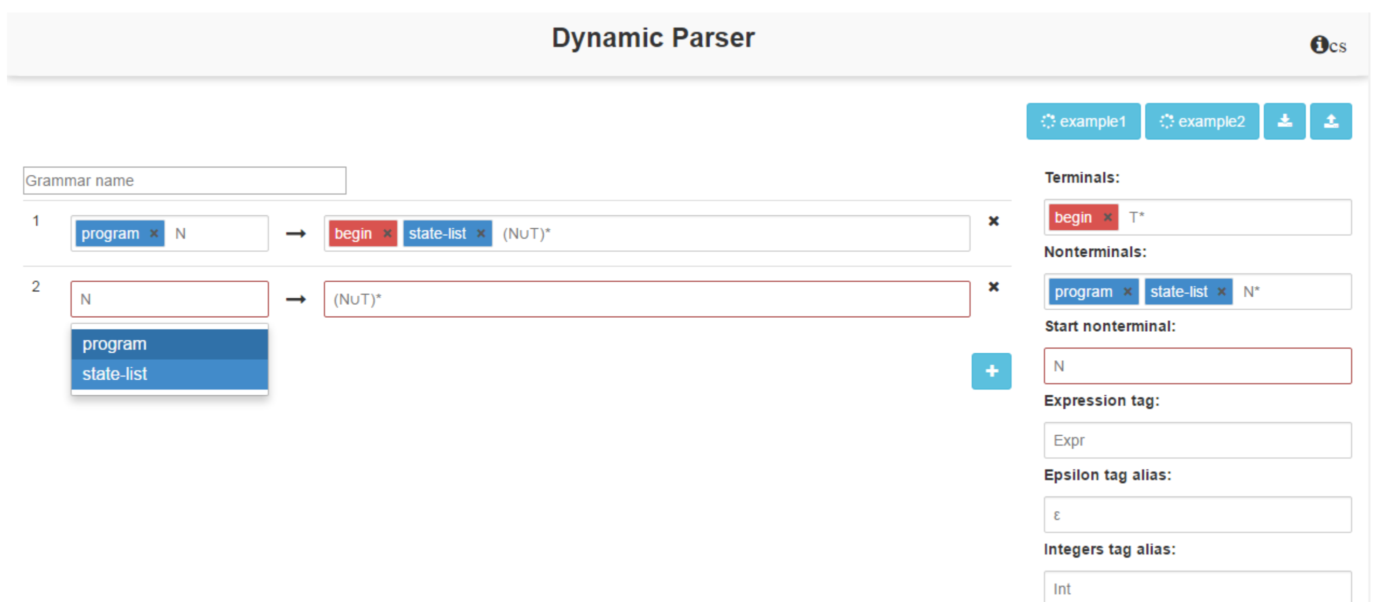
1. Text diplomové práce ve formátu PDF
2. Zdrojový text práce ve formátu ODT (docx)
3. Zdrojové soubory aplikace (manuál viz. kapitola 7.3)

Příloha B

Obrázek 12: Výchozí vzhled aplikace po načtení stránky



Obrázek 13: Ukázka volby typu nového symbolu při psaní pravidel



Obrázek 14: Ukázka vytváření pravidel - nabídka již existujících tagů (neterminálů) pro pravou stranu pravidla

Dynamic Parser

example1 example2

Example 1 (from Thesis, only top->down)

1 prog N → begin state (N∪T)*

2 state N → cmd item ; state (N∪T)*

3 state N → end (N∪T)*

4 cmd N → print (N∪T)*

5 cmd N → read (N∪T)*

6 cmd N → ε (N∪T)*

7 item N → int (N∪T)*

8 item N → id (N∪T)*

Terminals: begin ; end print read ε int id $ T*

Nonterminals: prog state cmd item N*

Start nonterminal: prog N

Expression tag: Expr

Epsilon tag alias: ε ε

Integers tag alias: int Int

Floats tag alias: Float

Strings tag alias: (e.g. "text") String

Identifier tag alias: id Id

LL Table:

	begin	;	end	print	read	int	id	$
prog	1							
state			3	2	2	2	2	
cmd				4	5	6	6	
item						7	8	

Generate LL Table

Input code:

```
begin
   42;
end
// end of code
```

Analyze input

Input in tokens:

1 begin

2 42 ;

3 end

4 // end of code

Syntax analysis result: ✔

Sequence of applied rules:

1
2
6
7
3

Dynamic Parser © Copyright 2017 Jaroslav Handlíř, FIT VUT Brno
xhandl05@stud.fit.vutbr.cz

Obrázek 15: Ukázka vzorového příkladu - kompletní gramatika, sestavená LL tabulka a provedená analýza