
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA GRAMATICKÝCH SYSTÉMECH

**PARSING BASED ON GRAMMAR SYSTEMS**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **ADAM SEDMÍK**
**AUTHOR**

**VEDOUCÍ PRÁCE** **prof. RNDr. ALEXANDER MEDUNA, CSc.**
**SUPERVISOR**



**BRNO 2022**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

# Zadání bakalářské práce


Student: **Sedmík Adam**
Program: Informační technologie
Název: **Syntaktická analýza založená na gramatických systémech**
**Parsing Based on Grammar Systems**
Kategorie: Teoretická informatika

Zadání:

1. Seznamte se podrobně s gramatickými systémy a jejich vlastnostmi dle pokynů vedoucího. Zaměřte se na systémy, jejichž komponenty mají využití v syntaktické analýze, např. LL či precedenční komponenty.
2. Studujte vlastnosti gramatických systémů v předchozím bodě dle pokynů vedoucího.
3. Dle pokynů vedoucího uvažujte alespoň 6 syntaktických struktur, včetně struktur, které nejsou bezkontextové. Popište jejich syntaktickou analýzu založenou na systémech z bodu 2.
4. Aplikujte a implementujte syntaktickou analýzu navrženou v předchozím bodě. Testujte ji.
5. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

1. Meduna, A.: Automata and Languages, Springer, London, 2000
2. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1 - 3, Springer, 1997, ISBN 3-540-60649-1
3. Aho, A. V., Sethi, R., Ullman, J. D.: Compilers : principles, techniques, and tools, Addison-Wesley, 2nd ed., 2007, ISBN 0321486811

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 26. října 2021

# Abstrakt

Práce se zabývá problematikou gramatických systémů a syntaktické analýzy. V práci jsou představeny kooperačně distribuované a paralelně komunikující gramatické systémy. Na základě znalostí o gramatických systémech je navrhnut nový typ gramatických systémů se zaměřením na modularizaci syntaktické analýzy. Jsou předvedeny metody syntaktické analýzy, metoda rekurzivního sestupu a precedenční syntaktická analýza. Navržené gramatické systémy jsou předvedeny na syntaktické analýze vlastního programovacího jazyka.



# Abstract

This thesis is focused on grammar systems and syntax analysis. Thesis introduces cooperating distributed and parallel communicating grammar systems. Based on knowledge of grammar systems, new grammar system is introduced. This grammar system focuses on modularization of syntax analysis. Shown are two methods of syntax analysis, recursive descent parsing and precedence parsing. Grammar systems introduced are demonstrated on syntax analysis of custom programming language.



# Klíčová slova

syntaktická analýza, gramatické systémy, překladač, rekurzivní sestup, bezkontextová gramatika

# Keywords

syntax analysis, grammar systems, compiler, recursive descent parsing, context-free grammar



# Citace

SEDMÍK, Adam. *Syntaktická analýza založená na gramatických systémech.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexander Meduna, CSc.

# Syntaktická analýza založená na gramatických systémech

## Prohlášení

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

.....
Adam Sedmík
11. května 2022

## Poděkování

Chtěl bych poděkovat vedoucímu práce, panu prof. RNDr. Alexandru Medunovi CSc., za trpělivost a odbornou pomoc při vypracování práce.

# Obsah

# Kapitola 1

# Úvod

Práce se zabývá teorií formálních jazyků, které se v moderní době využívají k analýze programovacích jazyků. V kapitole 2 jsou uvedeny základní pojmy z oblasti formálních jazyků, jako abeceda, řetězce, jazyky a gramatiky. Gramatikám je věnovaná podstatná část práce, se zaměřením na typ gramatik bezkontextových, které mají největší užití právě v rámci analýzy programovacích jazyků. Dále jsou popsány zásobníkové automaty, které jsou jedním z modelů pro bezkontextové gramatiky.

Na gramatiky navazují gramatické systémy, kde jejich myšlenka se spočívá v modularizaci a paralelizaci gramatik. V gramatických systémech může více gramatik tvořit výsledný jazyk systému, jednotlivé gramatiky tedy spolupracují. Gramatické systémy se dělí na dva hlavní typy. Prvním z nich jsou CD (Cooperating Distributed) gramatické systémy, ve kterých se jednotlivé gramatiky v rámci spolupráce střídají. Druhým typem jsou PC (Parallel Communicating) gramatické systémy, kde gramatiky pracují paralelně a v rámci synchronizace vyměňují svoje výsledky. Tyto gramatické systémy jsou popsány v kapitole 3.

Práce je zaměřena na návrh nového typu gramatických systémů. Tento gramatický systém je typu CD gramatických systémů. Narozdíl od klasických CD gramatických systémů, kde se jednotlivé gramatiky nácházejí na stejné úrovni, navržený systém třídí jednotlivé gramatiky do hierarchie, kde dochází k zanořování gramatik do dalších gramatik. Navržený gramatický systém je popsán v kapitole 4.

V kapitole 5 je uveden základ do syntaktické analýzy. Syntaktická analýza se používá ke kontrole správnosti zápisu programovacích jazyků. Na popis struktury programovacích jazyků se používají zmíněné bezkontextové gramatiky. V kapitole jsou popsány základní metody syntaktické analýzy, které jsou následně aplikovány na navržené gramatické systémy. V kapitole 6 je navržen jednoduchý programovací jazyk a popis implementace syntaktické analýzy podle navržených metod.

# Kapitola 2

# Teorie

Tato kapitola se zabývá základními pojmy z oblasti formálních jazyků, které jsou prerekvizitou této práce. Zmíněné a další definice lze nálezt v [3] a [4].

## 2.1 Základní definice

**Definice 2.1.1.** Abeceda je konečná, neprázdná množina elementů, které nazýváme symboly.

Abecedu můžeme značit symbolem $\Sigma$, následně symboly v abecedě značíme $\Sigma = \{a, b, c\}$

**Definice 2.1.2.** Řetězec je konečná posloupnost symbolů abecedy. Nechť $\Sigma$ je abeceda:

- $\varepsilon$ je řetězec nad abecedou $\Sigma$, $\varepsilon$ značí prázdný řetězec, neobsahuje žádný symbol
- pokud $x$ je řetězec nad $\Sigma$ a symbol $a \in \Sigma$ , potom $xa$ je řetězec nad abecedou $\Sigma$
- $\Sigma^*$ označuje všechny řetězce nad abecedou $\Sigma$

Mějme abecedu $\Sigma = \{a, b, c\}$ pak $aabc$ je jeden z řetězců nad abecedou $\Sigma$.

**Definice 2.1.3.** Délka řetězce je celkový počet symbolů v řetezci. Nechť x je řetězec nad abecedou $\Sigma$, Délka řetězce $x$, $|x|$, je definována:

- pokud $x = \varepsilon$, pak $|x| = 0$
- pokud $x = a_1 \dots a_n$, pak $|x| = n$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1, \dots, n$

Mějme řetězec $x = aabc$, pak $|x| = 4$

**Definice 2.1.4.** Nechť $x$ a $y$ jsou dva řetězce nad abecedou $\Sigma$. Konkatenace $x$ a $y$ je řetězec $xy$.

- Konkatenace prázdného řetězce : $x\varepsilon = \varepsilon x = x$

Mějme dva řetězce $x = abc$ a $y = acc$, konkatenace $xy = abcacc$

**Definice 2.1.5.** Nechť $x$ je řetězec nad abecedou $\Sigma$. Pro $i \geq 0$ je mocnina řetězce:

- $x^0 = \varepsilon$
- pro $x \geq 1$, $x^i = xx^{i-1}$

Mějme řetězec $x = abc$, pak $x^2 = abcabc$

**Definice 2.1.6.** Nechť $x = a_1 \dots a_n$ je řetězec. Pak reverzace řetězce $reversal(x) = a_n \dots a_1$

- $reversal(\varepsilon) = \varepsilon$

Mějme řetězec $x = abc$, pak $reversal(x) = cba$

**Definice 2.1.7.** Nechť $x$ a $y$ jsou řetezce nad abecedou $\Sigma$, $x$ je prefix $y$, pokud existuje řetězec $z$ nad $\Sigma$, kde platí $xz = y$

Mějme řetězec $abc$, $a$ je prefix

**Definice 2.1.8.** Nechť $x$ a $y$ jsou řetezce nad abecedou $\Sigma$, $x$ je suffix $y$, pokud existuje řetězec $z$ nad $\Sigma$, kde platí $zx = y$

Mějme řetězec $abc$, $c$ je suffix

**Definice 2.1.9.** Nechť $x$ a $y$ jsou řetezce nad abecedou $\Sigma$, $x$ je podřetězec $y$, pokud existují řetězce $z$ a $z'$ nad $\Sigma$, kde platí $zxz' = y$

- Prefix i suffix jsou podřetězce $y$

Mějme řetězec $abc$, $b$ je podřetězec

## 2.2 Jazyky

**Definice 2.2.1.** Nechť $\Sigma$ je abeceda a $L \subseteq \Sigma^*$, $L$ je jazyk nad abecedou $\Sigma$.

- Jazyk $L$ je konečný, pokud obsahuje konečný počet řetězců, jinak je nekonečný.

**Definice 2.2.2.** Nechť $L_1$ a $L_2$ jsou dva jazyky nad abecedou $\Sigma$. Sjednocení jazyků je definováno:

- $L_1 \cup L_2 = \{x : x \in L_1 \text{ nebo } x \in L_2\}$

**Definice 2.2.3.** Nechť $L_1$ a $L_2$ jsou dva jazyky nad abecedou $\Sigma$. Průnik jazyků je definován:

- $L_1 \cap L_2 = \{x : x \in L_1 \text{ a } x \in L_2\}$

**Definice 2.2.4.** Nechť $L_1$ a $L_2$ jsou dva jazyky nad abecedou $\Sigma$. Rozdíl jazyků je definován:

- $L_1 - L_2 = \{x : x \in L_1 \text{ a } x \notin L_2\}$

**Definice 2.2.5.** Nechť $L$ je jazyk nad abecedou $\Sigma$. Doplněk jazyka $L$ je definován:

- $\overline{L} = \Sigma^* - L$

**Definice 2.2.6.** Nechť $L_1$ a $L_2$ jsou dva jazyky nad abecedou $\Sigma$. Konkatenace jazyků je definována:

- $L_1L_2 = \{xy : x \in L_1 \text{ a } y \in L_2\}$

**Definice 2.2.7.** Nechť $L$ je jazyk nad abecedou $\Sigma$. Reverzace jazyka $L$ je definována:

- $reversal(L) = \{reversal(x) : x \in L\}$

**Definice 2.2.8.** Nechť $L$ je jazyk nad abecedou $\Sigma$. Pro $i \geq 0$ je mocnina jazyka:

- $L^0 = \{\varepsilon\}$
- pro $i \geq 1$, $L^i = LL^{i-1}$

**Definice 2.2.9.** Nechť $L$ je jazyk nad abecedou $\Sigma$. Iterace jazyka $L$ je definována:

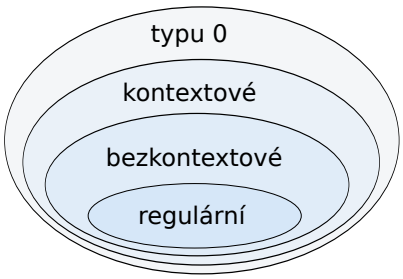
- $L^* = \bigcup_{i=0}^{\infty} L^i$
- pozitivní iterace, $L^+ = \bigcup_{i=1}^{\infty} L^i$
- $L^* = L^+ \cup L_0 = L^+ \cup \{\varepsilon\}$

## 2.3 Gramatiky

**Definice 2.3.1.** Gramatika je čtveřice $G = (N, T, S, P)$, kde:

- N je abeceda neterminálů
- T je abeceda terminálů, $N \cap T = \emptyset$
- S je počáteční neterminál, $S \in N$
- P je konečná množina pravidel nad $N \cup T$

Terminály jsou symboly, které může obsahovat výsledný řetězec. Neterminály slouží k zápisu pravidel a jsou to symboly, které se nenachází ve výsledném řetězci, ale podílí se na jeho postupné tvorbě. Gramatiky vycházejí z počátečního neterminálu a postupně aplikují pravidla, dokud není výsledný řetězec tvořen pouze terminály. Gramatiky se člení do tříd na základě formy pravidel, které povolují. Nejpoužívanějším členěním je Chomského hierarchie, znázorněná na obrázku 2.1.




Obrázek 2.1: Chomského hierarchie. Obrázek převzat z [1].

**Definice 2.3.2.** Formy pravidel na základě Chomského hierarchie jsou definovány:

- Pro neomezené gramatiky (typu 0), P je konečná množina pravidel tvaru $\alpha \to \beta$, kde $\alpha \in (N \cup T)^*$ a $\beta \in (N \cup T)^*$

- Pro kontextové gramatiky (typu 1), P je konečná množina pravidel tvaru $\alpha \to \beta$, kde $\alpha \in xNy$ a $\beta \in x(N \cup T)^* y$, kde $x, y \in (N \cup T)^*$
- Pro bezkontextové gramatiky (typu 2), P je konečná množina pravidel tvaru $\alpha \to \beta$, kde $\alpha \in N$ a $\beta \in (N \cup T)^*$
- Pro regulární gramatiky (typu 3), P je konečná množina pravidel tvaru $\alpha \to \beta$ , kde $\alpha \in N$ a $\beta \in T^*$ nebo $(T^*N)$

**Příklad 2.3.1.** *Mějme bezkontextovou gramatiku:*

- $G = (\{S\}, \{a, b\}, P, S)$
  - $P = \{S \to aSb, S \to ab\}$

*Tato gramatika může opakovaně aplikovat první pravidlo do doby, než je aplikované druhé pravidlo. Jakmile je aplikované druhé pravidlo, řetězec už neobsahuje žádné neterminály a je považován za výsledný. Z pravidel jde vidět, že gramatika na levé straně generuje symboly a, a na pravé straně stejný počet symbolů b. Jazyk generovaný gramatikou G lze zapsat* $L(G) = \{a^n b^n : n \geq 1\}$.

## 2.4 Zásobníkový automat

Zásobníkový automat je jeden z modelů pro bezkontextové gramatiky. Ke každé bezkontextové gramtice lze vymyslet zásobníkový automat, který rozpoznává její jazyk. Zasobníkový automat je stavové řízení, které kromě stavů ma také k dispozici jednoduchou paměť ve formě zásobníku.

**Definice 2.4.1.** Zásobníkový automat je sedmice $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, kde:

- Q je konečná množina stavů
- $\Sigma$ je vstupní abeceda
- $\Gamma$ je zásobníková abeceda
- $\delta$ je přechodová funkce tvaru $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$
- $q_0$ je počáteční stav, $q_0 \in Q$
- $z_0$ je počáteční symbol na zásobníku, $z_0 \in \Gamma$
- F je množina koncových stavů, $F \subseteq Q$

Zásobníkový automat začíná v počátenčním stavu s počátečním symbolem na zásobníku. Následně podle přechodových funkcí, které definují přechod podle aktualního stavu, symbolu na vstupu a symbolu na vrcholu zásobníku se provede přechod mezi stavy. Při přechodu se přečte následující symbol na vstupu a provede nahrazení symbolu na vrcholu zásobníku za symboly jiné. Zásobníkový automat pokračuje v přechodech, dokud neskončí v některém z koncových stavů a má přečtený celý vstup, nebo nemůže provést žádný přechod a skončí neúspěšně.

# Kapitola 3

# Gramatické systémy

Gramatické systémy reprezentují systém, kde více gramatik může společně generovat jazyk. Gramatické systémy obsahují dílčí gramatiky, které spolupracují na výsledné větné stavbě. V rámci práce jsou uvedeny hlavní typy gramatických systému a to CD (Cooperating Distributed) gramatické systémy a PC (Parallel Communicating) gramatické systémy. Informace o gramatických systémech jsou z prezentací předmětu TID[5][6].

## 3.1 CD gramatické systémy

Kooperačně distribuované gramatické systémy fungují sekvenčně. Při přepisování je aktivní pouze jedna z dílčích gramatik systému, která pracuje na větné formě. Dochází zde k přepínání mezi aktivní gramatikou systému podle určeného komunikačního protokolu.

**Definice 3.1.1.** CD gramatický systém je n–tice, $n \geq 1$, $\Gamma = (N, T, S, P_1, \ldots, P_n)$, kde:

- N je abeceda neterminálů
- T je abeceda terminálů, $N \cap T = \emptyset$
- S je počáteční neterminál, $S \in N$
- $P_1, \ldots, P_n$ jsou konečné množiny přepisovacích pravidel nad $N \cup T$, nazývané komponenty

### 3.1.1 Derivační módy

Derivačním módem se rozumí způsob, jakým si jednotlivé komponenty předávají řízení. Derivační módy také určují jak dlouho je jednotlivá komponenta aktivní.

#### Normální mód

V normálním módu můžou komponenty předávat řízení po libovolném počtu provedených pravidel. Lze tak konstatovat, že všechna pravidla se mohou aplikovat kdykoli nezávisle na komponentě. Normální mód tedy nepřináší žádné speciální vlastnosti oproti gramatice, jejíž pravidla jsou sjednocením pravidel komponent tohoto gramatického systému.

### Terminační mód

V terminálním módu pracuje každá komponenta dokud může. Aplikuje tedy přepisovací pravidla do té doby, než už žádné nemůže použít. Jakmile komponenta nemůže žádné pravidlo použít, nastane změna komponenty. Derivace proběhne úspěšně, nebo nastane situace, kdy žádná komponenta nemůže použít ani jedno pravidlo a derivace skončí neúspěšně.

**Definice 3.1.2.** Pro každé $i \in \{1, \ldots, n\}$ ukončující derivace í-té komponenty $x \ {}_i \Rightarrow^t y$ právě tehdy když:

- $x \Rightarrow^* y \in G_i = (N, T, S, P_i)$
- $y \not\Rightarrow z$ pro všechna $z \in (N \cup T)^*$

$G_i = (N, T, S, P_i)$ značí í-tou gramatiku a $y \not\Rightarrow z$ označuje, že nelze derivovat $y$ na $z$

### Mód právě k kroků

V tomto módu musí každá komponenta provést právě $k$ pravidel. Ve chvíli, kdy komponenta provede právě $k$ pravidel, předá řízení jiné komponentě. Pokud komponenta nemůže provést právě $k$ pravidel, derivace skončí neúspěšně.

**Definice 3.1.3.** Pro každé $i \in \{1, \ldots, n\}$ ukončující derivace í-té komponenty $x \ {}_i \Rightarrow^{=k} y$ právě tehdy když:

- $x \Rightarrow^k y \in G_i = (N, T, S, P_i)$

$x \Rightarrow^k y$ značí, že v gramatice proběhne $k$ derivací pro derivaci řetězce $x$ na $y$

### Mód nejvíce k kroků

Zde komponenta může provést maximálně $k$ pravidel, než je předáno řízení další komponentě. Komponenta tedy může předat řízení kdykoli, pokud počet provedených pravidel je menší nebo rovno jako $k$. Jakmile je počet provedených pravidel rovno $k$ komponenta předat řízení jiné komponentě musí. Derivace může skončit neúspěšně, pokud žádná komponenta nemůže provést ani jedno pravidlo a výsledný řetězec není zcela derivován.

**Definice 3.1.4.** Pro každé $i \in \{1, \ldots, n\}$ ukončující derivace í-té komponenty $x \ {}_i \Rightarrow^{\leq k} y$ právě tehdy když:

- $x \Rightarrow^j y \in G_i = (N, T, S, P_i)$ pro $j \leq k$

### Mód nejméně k kroků

Zde musí komponenta provést nejméně $k$ pravidel, aby mohla předat řízení. Pokud komponenta provedla alespoň $k$ pravidel, může předat řízení, nebo pokračovat dále. Pokud ale komponenta nemůže provést alespoň $k$ pravidel, derivace skončí neúspěšně.

**Definice 3.1.5.** Pro každé $i \in \{1, \ldots, n\}$ ukončující derivace í-té komponenty $x \ {}_i \Rightarrow^{\geq k} y$ právě tehdy když:

- $x \Rightarrow^j y \in G_i = (N, T, S, P_i)$ pro $j \geq k$

### 3.1.2 Generovaný jazyk

Jazyk generovaný CD gramatickými systémy je dán nejen derivacemi z počátečního neterminálu za použití pravidel komponent, ale i závisí také na použitém derivačním módu. Využitím jiného derivačního módu lze generovat jiný jazyk bez změny gramatického systému.

**Příklad 3.1.1.** *Mějme gramatický systém:*

- $\Gamma = (\{S, A, A', B', B\}, \{a, b, c\}, S, P_1, P_2)$
  - $P_1 = \{S \to S, S \to AB, A' \to A, B' \to B\}$
  - $P_2 = \{A \to aA'b, B \to cB', A \to ab, B \to c\}$

*V případě terminačního módu vznikne jazyk* $L_t(\Gamma) = \{a^n b^n c^m : m, n \geq 1\}$*, kde se sice komponenty* $P_1$ *a* $P_2$ *střídají, ale jakmile se provede třet, nebo čtvrté pravidlo z* $P_2$*, může dojít k situaci, kdy se budou nadále generovat pouze terminály ab nebo c. Naopak při použití módu* $k = 2$ *vznikne jazyk* $L_{k=2}(\Gamma) = \{a^n b^n c^n : n \geq 1\}$*, kde při použití třetího nebo čtvrtého pravidla z* $P_2$ *se musí použít obě dvě tato pravidla jinak při změně do* $P_1$ *nelze splnit podmínku, že se provedou práve dvě pravidla a tudíž je derivace neúspěšná.*

Například za pomocí pumping lemma[1] je možné dokázat, že jazyk $L_{k=2}(\Gamma)$ z příkladu není bezkontextový. S pomocí správně zvoleného derivačního módu se jazyk ale povedlo vygenerovat za použití bezkontextových pravidel. Z toho vyplývá, že s patřičně zvoleným derivačním módem a bezkontextovými pravidly, lze vytvořit gramatický systém, který je silnější než bezkontextové gramatiky.

[1] https://en.wikipedia.org/wiki/Pumping_lemma_for_context-free_languages

## 3.2 PC gramatické systémy

Paralelně komunikující gramatické systémy narozdíl od kooperačně distributivních gramatických systémů, ve kterých si komponenty předávají řízení mezi sebou, tak zde každá komponenta pracuje samostatně synchronizovaně na svých derivacích. Informace o derivovaných větách si předají až v momentě, kdy přijde na řadu komunikační krok.

**Definice 3.2.1.** PC gramatický systém je n–tice, $n \geq 1, \Gamma = (N, T, K, (P_1, S_1), \ldots, (P_n, S_n))$, kde:

- N je abeceda neterminálů
- T je abeceda terminálů, $N \cap T = \emptyset$
- K je množina dotazovacích symbolů, $K = \{Q_1, \ldots, Q_n\}$, $N \cap T \cap K = \emptyset$
- $S_1, \ldots, S_n$ jsou počáteční neterminály jednotlivých komponent
- $P_1, \ldots, P_n$ jsou konečné množiny přepisovacích pravidel komponent nad $N \cup T \cup K$

Odvození věty funguje následovně. V každém kroku provede každá komponenta jeden derivační krok podle svých pravidel, Pokud některá komponenta obsahuje dotazovací symbol, tak v daném kroku tato komponenta nahradí dotazovací symboly aktuální větou dotazovaných gramatik. Dotazované gramatiky nahradí svoji větnou formu počátečním neterminálem. Proběhl tedy komunikační krok. Speciálním pravidlem je, že pokud dotazovaná gramatika právě obsahuje také dotazovací symbol, nemůže se komunikační krok provést. Ukončení činnosti gramatického systému nástává ve chvíli, když daná komponenta (obvykle první) obsahuje pouze terminální symboly, bez ohledu na ostatní komponenty. Jazyk gramatického systému je tedy jazyk dané komponenty.

**Příklad 3.2.1.** *Mějme gramatický systém:*

- $\Gamma = (\{S_1, S_2\}\{a, b, c\}, \{Q_1, Q_2\}, (P_1, S_1), (P_2, S_2))$,
  - $P_1 = \{S_1 \to aS_1, S_1 \to aQ_2S_1, S_1 \to \epsilon, S_2 \to ca\}$
  - $P_2 = \{S_2 \to bS_2\}$

*Na tomto gramatickém systému si lze ukázat příklad derivace:* $(S_1, S_2) \Rightarrow (aS_1, bS_2) \Rightarrow (aaS_1, bbS_2) \Rightarrow (aaaQ_2S_1, bbbS_2) \Rightarrow (aaabbbS_2S_1, S_2) \Rightarrow (aaabbbcaS_1, bS_2) \Rightarrow (aaabbbcaaQ_2S_1, bbS_2) \Rightarrow (aaabbbcaabbS_2S_1, S_2) \Rightarrow (aaabbbcaabbS_2aS_1, bS_2) \Rightarrow (aaabbbcaabbS_2aaQ_2S_1, bbS_2) \Rightarrow (aaabbbcaabbS_2aabbS_2S_1, S_2) \Rightarrow (aaabbbcaabbS_2aabbS_2, bS_2) \Rightarrow (aaabbbcaabbS_2aabbac, bbS_2) \Rightarrow (aaabbbcaabbcaaabbca, bbS_2) = (a^3b^3ca^2b^2ca^3b^2ca, bbS_2)$. *Příklad byl uveden pro ukázku derivace, ale jazyk systému lze uvést jako poslupnost* $(a^nb^n \vee a^{n+1}b^n)$ *oddělených symbolem c, kde n je různé v každém oddělení a věta je ukončena symboly* $ca^n$, *v případě, že nedojde ke komunikačnímu kroku, jazyk také obsahuje* $a^n$.

# Kapitola 4

# Navržené řešení

Navržený gramatický systém je stejného typu jako CD gramatické systémy. Jednotlivé komponenty pracují sekvenčně. Navržený systém obsahuje hlavní gramatiku, která jako komponenty používá gramatiky jiné a dochází zde k přepínání mezi hlavní gramatikou a komponenty. Jednotlivé komponenty mohou také obsahovat další komponenty. Z hlavní gramatiky se tvoří stromová struktura komponent, které si předávají řízení. Navržený gramatický systém má také podobnosti PC gramatickým systémům, kde každá komponenta derivuje svůj počáteční neterminál samostatně.

**Definice 4.0.1.** Mějme gramatiku $G_0 = (N, T, P, S, G_1 \dots G_n)$, kde

- $N$ je abeceda neterminálů
- $T$ je abeceda terminálů, $N \cap T = \emptyset$
- $S$ je počáteční neterminál
- $G_n$ je gramatika ve tvaru $G_n = (N, T, P, S, G_1 \dots G_n)$ - může obsahovat další gramatiky, $n \geq 0$
- $P$ je konečná množina přepisovacích pravidel[1] nad $N \cup T \cup z$, kde $z$ jsou počáteční neterminály gramatik $G_1, \dots, G_n$

*Jako vnořenou gramatiku můžeme nazývat gramatiku, která je komponentou jiné gramatiky. Nadřazená gramatika je gramatika, která jako komponentu používá právě tuto gramatiku.*

Jednotlivé gramatiky provádějí vlastní přepisovací pravidla, a pokud chtějí provést derivaci některého z počátečních terminálů vnořené gramatiky, provedou změnu gramatiky. Při změně gramatiky si vnořená gramatika provádí svoje derivace z vlastního počátečního neterminálu bez znalostí o derivacích nadřazené gramatiky. Stejně tak nadřazená gramatika nemá znalosti o vnořené gramatice, kromě informace zda derivace dopadla úspěšně. Jakmile proběhne úspěšná derivace vnořené gramatiky, provede se návrat zpět do nadřazené gramatiky, která může pokračovat ve svojí derivaci.

**Definice 4.0.2.** V rámci příkladů se bude značit $\Rightarrow$ derivace v rámci vlastní gramatiky a $\Rightarrow_{G_n}$ přechod mezi gramatikami.

[1] Pravidla mohou být kteréhokoliho typu. Práce se soustředí na pravidla bezkontextová.

**Příklad 4.0.1.** *Mějme gramatiky:*

- $G_0 = (\{A, S_0\}, \{a, b\}, P, S_0, G_1)$
  - $P = \{S_0 \to AS_1, A \to aAb, A \to ab\}$
- $G_1 = (\{S_1\}, \{c\}, P, S_1)$
  - $P = \{S_1 \to S_1c, S_1 \to c\}$

*Výsledný jazyk generovaný tímto systémem je* $L_0 = \{a^n b^n c^m : m, n \geq 1\}$. *Nejdříve se použije první pravidlo z* $G_0$, *následně se může použít libovolný počet přepisů druhého pravidla, dokud se nepoužije třetí pravidlo a ukončí se přepis v* $G_0$. *Následně zbývá neterminál* $S_1$, *kde se provede přechod mezi gramatikami a* $G_1$ *začne derivovat neterminál* $S_1$ *do řetězce* $c^m$. *Jakmile skončí s derivací, provede se přechod zpět do nadřazené gramatiky* $G_0$, *kde výsledný řetězec nahradí neterminál* $S_1$. *V gramatice* $G_0$ *se už nenachází žádný další neterminál a derivace se může ukončit s výsledným řetězcem. Také zde nezávisí na pořadí, jestli se nejdříve budou aplikovat pravidla v* $G_0$, *nebo se přejde do gramatiky* $G_1$, *tyto operace jsou nezávislé a mohou se kombinovat bez vlivu na výsledek, možno i provést jen některá pravidla z* $G_0$, *provést změnu gramatiky a po navrácení dokončit derivaci.*

## 4.1 Jazyk

Jednotlivé gramatiky mohou pracovat nezávisle - jejich abecedy jsou tedy oddělené a při změně gramatiky dojde i ke změně abecedy. Lze získat abecedu celého systému sjednocením abeced jednotlivých gramatik. Výsledná abeceda je tedy abecedou celého systému. Pokud gramatika obsahuje počáteční neterminál vnořené gramatiky, její vlastní výsledný řetězec je řetězec bez těchto neterminálů, a jakmile vnořená gramatika provede nahrazení neterminálu, výsledný řetězec je řetězec celého systému. Při nahrazení řetezcem, může být tento řetězec jakýkoli řetězec z vnořené gramatiky. Proto je potřebné uvést následující definici:

**Definice 4.1.1.** Nechť $L(G)$ je jazyk generován gramatikou $G$, $L(G)_\Sigma$ značí jakýkoli řetězec z jazyka $L(G)$.

**Příklad 4.1.1.** *Mějme gramatiky:*

- $G_0 = (\{A, B, S_0\}, \{b\}, P, S_0, G_1, G_2)$,
  - $P = \{S_0 \to AS_0B, S_0 \to AB, A \to S_1, B \to S_2b\}$
- $G_1 = (\{S_1\}, \{a\}, P, S_1)$
  - $P = \{S_1 \to aaa\}$
- $G_2 = (\{C, D, S_2\}, \{c, d\}, P, S_2)$
  - $P = \{S_2 \to Cd, C \to cD, C \to c, D \to dD, D \to d\}$

*Gramatika* $G_2$ *generuje jazyk* $L_2(G_2) = \{cd^n : n \geq 1\}$
*Gramatika* $G_1$ *generuje jazyk* $L_1(G_1) = \{aaa\} = \{a^3\}$
*Gramatika* $G_0$ *generuje vlastní jazyk* $L_0(G_0) = \{S_1^n(S_2b)^n : n \geq 1\} = \{b^n : n \geq 1\}$

*Dále je uveden příklad derivace. Přeškrtlé symboly značí symboly, které právě aktivní gramatika nemá ve svém rámci:*
$S_0 \Rightarrow AS_0B \Rightarrow S_1S_0B \Rightarrow_{G_1} S_1\cancel{S_0B} \Rightarrow aaa\cancel{S_0B} \Rightarrow_{G_0} \cancel{aaa}S_0B \Rightarrow \cancel{aaa}ABB \Rightarrow \cancel{aaa}S_1BB \Rightarrow_{G_1} \cancel{aaa}S_1\cancel{BB} \Rightarrow \cancel{aaa}aaa\cancel{BB} \Rightarrow_{G_0} \cancel{aaaaaa}BB \Rightarrow \cancel{aaaaaa}S_2bB \Rightarrow \cancel{aaaaaa}S_2bS_2b \Rightarrow_{G_2} \cancel{aaaaaa}S_2\cancel{bS_2b} \Rightarrow \cancel{aaaaaa}Cdb\cancel{S_2b} \Rightarrow \cancel{aaaaaa}cdb\cancel{S_2b} \Rightarrow_{G_0} \cancel{aaaaaa}cdbS_2b \Rightarrow_{G_2} \cancel{aaaaaa}cdb\cancel{S_2b} \Rightarrow \cancel{aaaaaa}cdb\cancel{C}db \Rightarrow \cancel{aaaaaa}cdbcDd\cancel{b} \Rightarrow \cancel{aaaaaa}cdbcdd\cancel{b} \Rightarrow_{G_0} \cancel{aaaaaa}cdbcddb \ldots aaaaaacdbcddb$

*V jazyce $L_0$ nelze jednoduše nahradit $S_2$ za $cd^n$ pro získání jazyka celého systému, protože pokud se použije gramatika $G_2$ vícekrát, může se generovat v každém použití jiný řetězec (jak lze vidět v ukázce derivace). Při nahrazení by vznikl jazyk $L_{invalid}(G_0) = \{a^{3n}(cd^mb)^n : n \geq 1, m \geq 1\}$, do kterého výsledná derivace nepatří, jelikož v jazyku $L_{invalid}$ by mělo být $d^m$ pořád stejné pro každé $n$. Proto lze využít definici a výsledný jazyk zapsat jako $L(G_0) = \{S_1^n(S_2b)^n : n \geq 1\} = \{L_1(G_1)_\Sigma^n(L_2(G_2)_\Sigma b)^n : n \geq 1\}$, kde v rámci $n$ opakování $L_2(G_2)_\Sigma$ značí, že v každém opakování se může generovat jiný řetězec náležící do $L_2(G_2)$.*

## 4.2 Ekvivalence s klasickými gramatikami

Z definice navrženého gramatického systému lze pozorovat, že neterminály klasické gramatiky jsou nahrazeny vnořenými gramatikami a jejich počatečními neterminály. Navržený gramatický systém je tedy modularizace klasických gramatik. Systém navržený pro tuto práci umožnuje později vytvořit již spojený gramatický systém pro celou syntaktickou analýzu. Je možné ukázat, že navržený systém roven klasické gramatice, pomocí jeho tranformace.

**Příklad 4.2.1.** *Mějme gramatický systém s gramatikami $G_0, G_1$:*

- $G_0 = (\{A, S_0\}, \{a\}, P, S_0, G_1)$,
  - $P = \{S_0 \to AS_0, S_0 \to A, A \to S_1, A \to a\}$
- $G_1 = (\{A, S_1\}, \{b\}, P, S_1)$,
  - $P = \{S_1 \to AS_1, S_1 \to A, A \to b\}$

*V příkladu nelze jednoduše sjednotit abecedy a pravidla pro tvorbu jedné gramatiky, jelikož abeceda neterminálů se částečně překrývá a sjednocením by vznikl jiný jazyk. Překrytí se lze zbavit změnou abecedy jedné z gramatik bez vlivu na její výsledek.*

- $G_0 = (\{A, S_0\}, \{a\}, P, S_0, G_1)$,
  - $P = \{S_0 \to AS_0, S_0 \to A, A \to S_1, A \to a\}$
- $G_1 = (\{B, S_1\}, \{b\}, P, S_1)$,
  - $P = \{S_1 \to BS_1, S_1 \to B, B \to b\}$

*Nyní se je možné aplikovat sjednocení bez ovlivnění výsledku gramatik. Výsledná klasická gramatika tedy vypadá následovně:*

- $G_0 = (\{A, B, S_0, S_1\}, \{a, b\}, P, S_0)$,
  - $P = \{S_0 \to AS_0, S_0 \to A, A \to S_1, A \to a, S_1 \to BS_1, S_1 \to B, B \to b\}$

*Pokud je to možné, lze na výslednou gramatiku použít další zjednodušovací pravidla.*

## 4.3 Vlastnosti

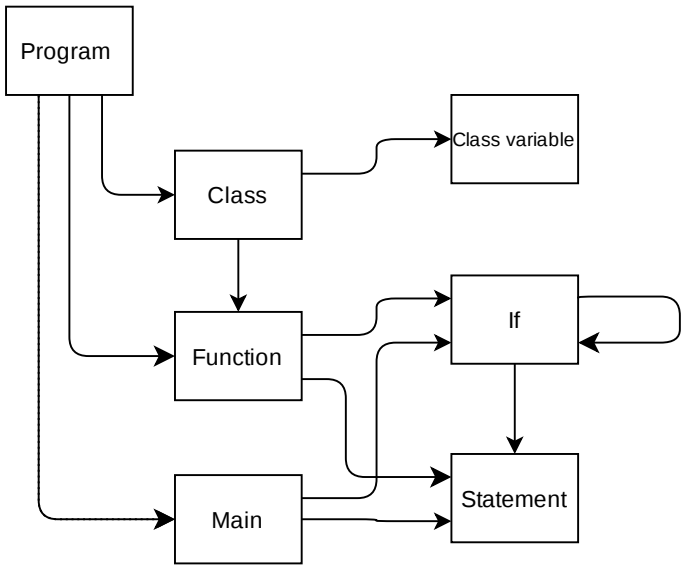
I přestože navržený gramatický systém se neliší funkcionálně od klasických gramatik, lze ho považovat za jiný pohled na gramatiky. Na každou gramatiku lze nahlížet jako na svoji vlastní strukturu, která je vložena do struktur jiných. Gramatika může považovat struktury vnořených gramatik jako její terminální symboly bez ohledu na její funkcionalitu. Vnořování gramatik také může reprezentovat různé struktury v rámci analýzy programovacích jazyků, kde se například v definici funkce může objevit podmínka do které se následně lze zanořit. Jako jedno ze speciálních pravidel vnoření lze považovat, že pokud počáteční neterminál vnořené gramatiky obsahuje i nadřazená gramatika, zde se může rozhodnout, zda chce aplikovat vnoření nebo si neterminál derivovat samostatně. Pokud se rozhodne o vnoření, může to značit, že se jedná o jinou strukturu, než kterou by daná gramatika generovala sama. Lze se podívat na jaké vlastnosti lze narazit, když se koukáme na gramatiky z tohoto pohledu.

### 4.3.1 Rekurze

Jednou z vlastností je rekurze, kdy do gramatiky vnoříme stejnou gramatiku, nebo gramatiku, která je jí nadřazená. Takto by šlo v syntaktické analýze analyzovat struktury, které mohou obsahovat samy sebe. V progamovacích jazycích to jsou tedy například podmínky, které obsahují příkazy a další podmínky, nebo vnořování cyklů. Došlo by k analýze cyklu, ve kterém se analyzují příkazy a mezi těmito příkazy se nachází další cyklus, do kterého se vnoří a dojde ke změně gramatiky. V tomto případě změna gramatiky na tu stejnou, ale začínajíc znovu od počátečního neterminálu, která analyzuje cyklus bez znalosti o cyklech nadřazených. Po analýze vnořeného cyklu má nadřazená gramatika znalost o úspěšnosti analýzy vnořeného cyklu a může pokračovat v analýze svých příkazů. Pomocí navrženého gramatického systému si tedy lze strukturovat i takovéto rekurzivní modely. Takto navržené rekurzivní gramatické systémy by měly být schopné derivace i bez použití vnořené gramatiky (ukončovací podmínka), aby nenastala nekonečná rekurze.

### 4.3.2 Grafová reprezentace

Z pohledu na komponenty gramatiky jako na struktury lze gramatiky reprezentovat grafem. Z takového grafu můžeme získat informace o zanořování a udělat si přehled, jak jednotlivé komponenty budou mezi sebou pracovat. Každá gramatika generuje vlastní jazyk a vnoření do další gramatiky může být aplikováno vícekrát. Z grafu lze vidět jaké jazyky se mohou v rámci celého jazyka gramatiky generovat. Na obrázku 4.1 je zobrazena možná struktura progamu. Každý vrchol v obrázku reprezentuje gramatiku a hrany reprezentují možné vnoření.




Obrázek 4.1: Jednoduchá struktura programu

Program může obsahovat definici tříd, funkcí a metodu *main*, jak mohou být v programu poskládány záleží na gramatice *Program*. Gramatika může například vygenerovat více neterminálů do vnoření do gramatiky *Class*, kde se po vnoření zkontroluje správnost konstrukce třídy, kde dále může dojít ke vnoření pro kontrolu definice funkce třídy a atribut funkce. Gramatika *Program* i *Class* obsahují vnořenou gramatiku *Function*, tato gramatika je tedy identická pro obě možnosti vnoření a může se opětovně použít pro kontrolu funkcí programu i funkcí tříd. Po kontrole vnořených gramatik následuje návrat stejnou cestou po opačných směrech hran.

## 4.4 Srovnání s CD gramatickými systémy

I přestože bylo ukázáno, jak jsou navržené gramatiky ekvivalentní klasickým gramatikám, kooperačně distribuované gramatické systémy získavají svoji sílu podle derivačních módů. Jelikož navržené systémy jsou inspirací sekvenční funkcionalitou, lze si ukázat jak by mohly gramatiky změnit funkcionalitu ,pokud použijeme patřičné derivační módy.

### Terminační mód a normální mód

V těchto módech nenastává žádná změna, gramatické systémy mohou totiž použít vnoření kdykoli chtějí. Jediný rozdíl v terminačním módu je, že nadřazená gramatika musí nejdříve aplikovat všechna možná pravidla, než může provést vnoření. Tato skutečnost ale neovlivňuje funkctionalitu.

### Mód právě k kroků

Nadřazená gramatika musí provést právě $k$ kroků, po kterých následně musí proběhnout vnoření, pokud chce nadřazená gramatika prokračovat ve svých derivacích. Je zde garantován určitý počet vnoření do dalších gramatik, podle toho jak dlouho chce nadřazená gramatika provádět derivace.

### Mód nejvíce k kroků

Zde je gramatika méně limitována než v módu právě $k$ kroků, ale opět musí zajistit vnoření, pokud chce pokračovat v derivacích.

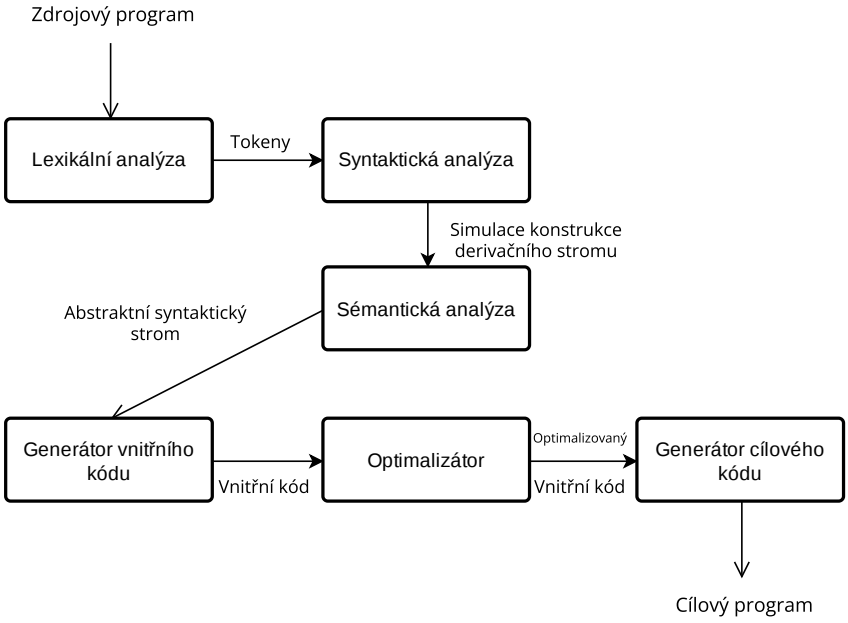
### Mód nejméně k kroků

Gramatika je omezena z opačného hlediska, kdy musí provádět derivace, aby bylo povoleno vnoření. Počet vnoření je omezen průběhem nadřazené gramatiky, kdy gramatika může předat řízení až poté, co proběhlo alespoň $k$ kroků.

# Kapitola 5

# Syntaktická analýza

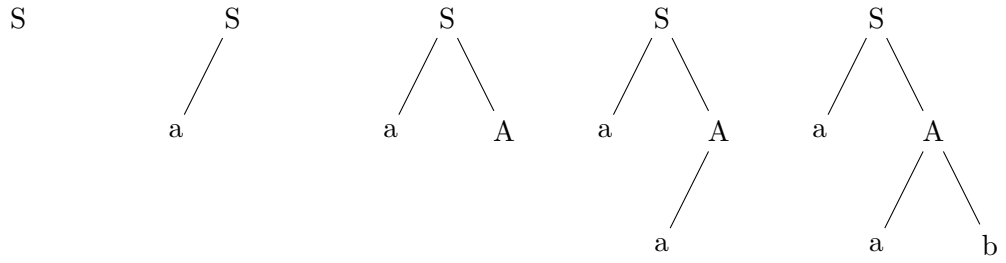
Syntaktická analýza je jedna z částí kompilátoru, programu, který převádí zdrojový jazyk do jazyka cílového. Kompilátor se skládá z několika základních částí. První z nich je lexikální analyzátor, který na vstupu dostává zdrojový text programovacího jazyka a převádí ho na tvz. tokeny. Tokeny jsou symboly, které reprezentují jednotlivé prvky daného jazyka. Tyto tokeny jsou posílány syntaktickému analyzátoru, který je používá ke kontrole syntaktické správnosti jazyka. Za pomocí gramatik kontroluje, zda tokeny přicházejí ve správném pořadí. Gramatiky také využívá k tvorbě výsledného derivačního stromu aplikací pravidel gramatiky. Tento strom se může tvořit dvěma způsoby a rozlišujeme syntaktickou analýzu shora dolů a zdola nahoru. Další část kompilátoru je sémantická analýza, která kontroluje datové typy, deklarace apod. Následně na základě syntaktické a sémantické analýzy probíhá překlad do vnitřního mezikódu, který je případně i optimalizován. Vnitřní kód je následně převeden do cílového kódu, čímž práce kompilátoru končí. O kompilátorech a jejich částech se lze dozvědět v publikaci [2]. Práce se věnuje lexikální a syntaktické analýze, ale schéma kompilátoru lze vidět na obrázku 5.1.




Obrázek 5.1: Schéma kompilátoru

## 5.1 Shora dolů

Syntaktická analýza shora dolů tvoří derivační strom (5.2) z počátečního neterminálu. Postupně na základě vstupního řetězce aplikuje pravidla gramatiky a tvoří tvz. levý rozbor gramatiky. Vstupní řetězec je porovnáván s aktuálním stavem gramatiky, kde terminální symboly by měly odpovídat vstupnímu řetězci.




Obrázek 5.2: Postup derivačního stromu syntaktické analýzy shora dolů

### 5.1.1 LL-gramatiky

LL-gramatika je specialní typ bezkontextové gramatiky užitečná pro syntaktickou analýzu shora dolů. LL-gramatiky jsou označovány $LL(k)$, kde $k$ je počet následujících symbolů, které je třeba znát k deterministickému rozboru věty. V LL-gramatice LL(1) tedy stačí znát jeden následující symbol, množina pravidel této gramatiky má tedy na pravé straně pravidel na začatku alespoň jeden terminál. Případně se může na pravé straně nacházet neterminál, pokud jeho derivace na začátku obsahuje alespoň jeden terminál. Aby byly gramatiky deterministické, musí také platit, že pro každou derivaci neterminálu se na pravé straně pravidel se daný symbol může vyskytnout maximálně jedenkrát.

### 5.1.2 Množina FIRST a LL tabulka

Jednoduchou syntaktickou analýzu lze vytvořit kontrukcí takzvané LL tabulky za pomocí množiny FIRST dané LL-gramatiky. Množina FIRST obsahuje všechny terminály pro daný neterminál, kterými lze řetězec derivovat.

**Definice 5.1.1.** Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Pro každé $x \in (N \cup T)^*$ je definováno $FIRST$(x) jako:

- $FIRST(x) = \{a : a \in T, x \Rightarrow^* ay; y \in (N \cup T)^*\}$

**Příklad 5.1.1.** *Mějme LL-gramatiku $G = (N, T, P, < prog >)$ s očíslovanými pravidly:*

- *1:*$< prog > \rightarrow begin[< stat >$
- *2:*$< stat > \rightarrow ]end$
- *3:*$< stat > \rightarrow id =< val >; < stat >$
- *4:*$< stat > \rightarrow function(); < stat >$
- *5:*$< val > \rightarrow num$

- *6:< val >→ string*

*Pro každý terminál jeho množina FIRST obsahuje právě tento terminál:*
*FIRST(begin) = {begin}, FIRST(]) = {]}, FIRST(id) = {id}, atd...*
*Do množin FIRST neterminálů tedy přidáváme první terminály podle jejich pravidel: < prog >→ begin[< stat >, do FIRST(<prog>) přidáme begin a následně podle všech pravidel. Získáme tedy množiny:*
*FIRST(<prog>) = {begin}*
*FIRST(<stat>) = {id, function, ]}*
*FIRST(<val>) = {num, string}*

Následně lze vytvořit LL-tabulku. Ta se skládá z řádků neterminálů a sloupců terminálů, kde průnik sloupce a řádku je pravidlo, které se má v případě derivace neterminálu použít. Průniky také značí, který terminál je právě zpracováván, a je tedy obsažen v množině FIRST neterminálu na daném sloupci. Pokud je průnik prázdný, nelze použít žádné pravidlo a derivace je neúspěšná. Nastane-li situace, že by v jednom průniku bylo více pravidel, tabulka není validní LL-tabulka a tedy gramatika také není validní LL-gramatika. Pro příklad lze vytvořit následující LL-tabulku:

| | begin | end | id | function | num | string | [ | ] | ( | ) | = | ; |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <prog> | 1 | | | | | | | | | | | |
| <stat> | | | 3 | 4 | | | | 2 | | | | |
| <val> | | | | | 5 | 6 | | | | | | |

### 5.1.3 Rekurzivní sestup

Rekurzivní sestup je implementace syntaktické analýzy využívající LL-tabulku. Každý neterminál je reprezentován procedurou, která kontroluje načítané symboly dané gramatiky. Dochází zde k zanořování na základě použitých pravidel. Proceduru pro neterminál s obecným pravidlem $X \to Y_1 \dots Y_n$ lze reprezentovat následujícím algoritmem (1):

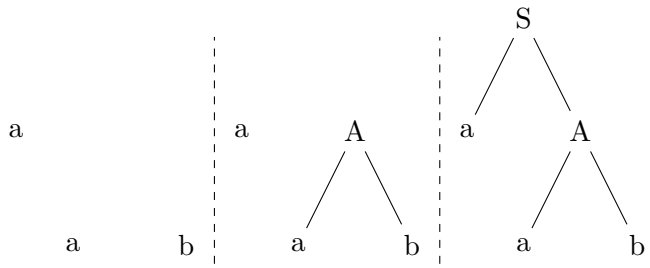
**Algoritmus 1** Rekurzivní sestup

```
1:  procedure Y
2:      for i = 1 to n do
3:          if X_i is nonterminal then
4:              call procedure X_i
5:          end if
6:          if X_i is terminal and is equal to current input symbol then
7:              read next input and continue
8:          else
9:              error
10:         end if
11:     end for
12:     success
13: end procedure
```

Výhodou metody rekurzivního sestupu je jednoduchá implementace. Naopak nevýhodou je, že každá procedura musí být implementována manuálně pro každé pravidlo a při částečné změně gramatiky musí být ovlivněné procedury reimplementovány.

## 5.2 Zdola nahoru

Syntaktická analýza zdola nahora pracuje opačně než syntakticá analýza shora dolů. Místo tvorby derivačního stromu od počátečního neterminálu, tvoří derivační strom (5.3) od terminálů nahoru a postupně skládá pravidla, dokud se nedostaně k počátečnímu neterminálu s přečteným vstupem. Je tedy tvořen tvz. pravý rozbor gramatiky.




Obrázek 5.3: Postup derivačního stromu syntaktické analýzy zdola nahoru

### 5.2.1 Precedenční syntaktická analýza

Jednou z metod pro syntaktickou analýzu zdola nahoru je precedeční syntaktická analýza. Tato metoda využívá zásobník pro zpracování vstupu a precedenční tabulku pro rozhodování podle aktualního stavu zásobníku a aktuálního vstupu. Precedenční tabulka obsahuje sloupce a řádky označeny terminály a speciálním symbolem označující konec vstupu. Sloupce označují symboly na vstupu a řádky symboly na vrcholu zásobníku. V průniku sloupců a řádků jsou speciální znaky $>, <, =$ značící, co s danými symboly dělat: redukce, posun a vložení na zásobník respektivně. Pokud je pole tabulky prázdné, značí to chybu v syntaktické analýze. Precedenční tabulka je vytvořena na základě pravidel ohledně precedence operátorů, asociativity, tvorby závorek a podobně. Po tvorbě precedenční tabulky pracuje syntaktická analýza podle následujícího algoritmu (2) :

**Algoritmus 2** Precedenční syntaktická analýza

```
1: procedure TOP
2:     returns terminal on stack closest to the top
3: end procedure
4: procedure PRECEDENCE PARSING
5:     push($) (bottom of stack and end of input symbol)
6:     repeat
7:         a = top
8:         b = current input symbol
9:         switch Table[a, b] do
10:            case =
11:                push(b) and read next input symbol
12:            case <
13:                exchange a for a < on top of stack
14:                push(b) and read next input symbol
15:            case >
16:                if < y is on top of stack and rule A → y exists then
17:                    exchange < y for A on top of stack
18:                else
19:                    error
20:                end if
21:            case empty
22:                error
23:    until b == $ and top == $
24:    success
25: end procedure
```

Algoritmus bude běžet dokud není prázdný vstup a na vrcholu zásobníku je pouze výsledný neterminál a symbol vrcholu zásobníku, nebo dokud algoritmus nevyhodí chybu. Dále je třeba implementovat patřičné redukce na základě pravidel gramatiky. Po tvorbě precedeční tabulky je precedeční syntaktická analýza jednoduchá na implementaci a užitečná pro zpracovávání závorkovaných výrazů.

## 5.3 Syntaktická analýza pro navržené gramatické systémy

Navržený gramatický systém je modulární. Syntaktická analýza na něm založená se bude zabývat moduly - jeden modul pro každou gramatiku v gramatickém systému. Moduly sdílí lexikální analyzátor a každý modul pracuje samostatně jakmile je volán. Přechod do modulu je dán předchozím modulem, který potřebuje použít vnořenou gramatiku. Pokud dojde ke změně modulu, předchozí modul si uchovává informace o své syntaktické analýze a pokračuje v ní po návratu do modulu zpět. V rámci rekurzivních gramatik může být daný modul volán vícekrát, každé volání je samostatná instance dáného modulu se svým vlastním stavem syntaktické analýzy. Modulární rozložení umožnuje použít pro každý modul jiný typ nebo metodu syntaktické analýzy, právě tu, která je pro danou gramatiku nejvhodnější.

# Kapitola 6

# Implementace

V této kapitole je navržen jednoduchý programovací jazyk a popsána implementace syntaktického analyzátoru pro navžený jazyk. Implementace obsahuje lexikální a syntaktickou analýzu pro navržené gramatické systémy. Navržený programovací jazyk je rozdělen do dílčích gramatik, které se střídají podle právě anlyzované části programu.

## 6.1 Technologie a architektura

Program je implementován v jazyce C++ s použitím standardních knihoven jazyka. Je obsažen Makefile pro sestavení a spuštění programu. Dále jsou obsaženy testovací soubory s příklady navrhnutého jazyka. Program obsahuje dědičnou hlavní třídu syntaktické analýzy, kterou dědí každý modul syntaktické analýzy a zde si implementuje modul svoji gramatiku. Lexikální analýza je sdílena v nadtřídě mezi všemi moduly. Po spuštění programu je spuštena hlavní nadřazená gramatika, která obstará postupné spuštění modulů gramatik dalších. Moduly mohou být spouštěny do doby než je přečten celý vstup a provede se návrat zpět do nadřazené gramatiky, která ukončí běh syntaktické analýzy. Po kontrole gramatiky daného modulu, se provede návrat do předchozího modulu nadřazené gramatiky, kterému předá informace o úspěšnosti kontroly gramatiky. Pokud nastane neúspěch syntaktické analýzu modulu, tato informace se pronese zpět do hlavní gramatiky, která ukončí běh progamu s neúspěchem. Jestliže se gramatika rozhoduje zda pokračovat ve své analýze nebo předá řízení rozhoduje se podle množin FIRST svých neterminálů a množin FIRST vnořených gramatik.

## 6.2 Navržený jazyk

Navržený jazyk je inpirací syntaxe jazyka C++, s rozdílem, že neobsahuje funkci *main*, ale umožňuje použít příkazy v jakékoli částí programu jako například jazyk Python. S jazykem Python má také společné, že není třeba definovat datový typ proměné. V následujících sekcích bude jazyk rozebrán po jednotlivých gramatikách a ukázána pravidla gramatiky, z kterých lze vidět struktura jazyka. Pro lepší pochopení jsou označeny $S_{subscript}$ počáteční neterminály vnořených gramatik a $symbol_t$ označení pro terminální symboly.

**Hlavní tělo programu**

- $S_{main} \rightarrow S_{class} S_{main}$
- $S_{main} \rightarrow S_{function} S_{main}$
- $S_{main} \rightarrow S_{condition} S_{main}$
- $S_{main} \rightarrow S_{statement} S_{main}$
- $S_{main} \rightarrow EOF_t$

V hlavním těle programu nedochází ke větší kontrole vstupního řetězce, pouze se rozhoduje zda se je na vstupu některá ze struktur jazyka, a patřičně předá řízení. Pokud není na vstupu žádná ze zadaných struktur skončí z chybou, pokud ale všechny struktury byly analyzovány a je konec vstupu analýza skončí úspěšně.

**Třídy**

- $S_{class} \rightarrow keyword\_class_t\ indentifier\_class_t\ \{_t\ BODY$
- $BODY \rightarrow id_t\ =_t\ VALUE\ ;_t\ BODY$
- $BODY \rightarrow S_{function}\ BODY$
- $BODY \rightarrow \}_t$
- $VALUE \rightarrow number_t$
- $VALUE \rightarrow string_t$

Třídy jsou zde pouze názvem, v navrženém jazyce nelze vytvořit více instancí jedné třídy. Třídy tedy spíše fungují jako proměné, které mohou obsahovat více atributů a funkce. Gramatika tříd kontroluje jejich správnou deklaraci, a následně tělo třídy, kde se mohou objevit atributy třídy a nebo může nastat vnoření do kontroly definice funkce.

**Funkce**

- $S_{function} \rightarrow keyword\_function_t\ indentifier\_function_t\ (_t\ PARAMS\ \{_t\ BODY$
- $PARAMS \rightarrow id_t\ PARAM$
- $PARAMS \rightarrow )_t$
- $PARAM \rightarrow ,_t\ id_t\ PARAM$
- $PARAM \rightarrow )_t$
- $BODY \rightarrow S_{function\ condition}\ BODY$
- $BODY \rightarrow S_{statement}\ BODY$
- $BODY \rightarrow S_{return}\ BODY$
- $BODY \rightarrow \}_t$

Gramatika pro funkce je společná pro tělo programu a pro třídy. Opět dochází ke kontrole deklarace funkce, s následnou kontrolou parametrů. V těle funkce pak dochází k zanořování podle struktury těla, kde se mohou objevit příkazy, podmínky nebo příkaz pro návrat z funkce.

**Return**

- $S_{return} \rightarrow keyword\_return_t\ VAR$
- $VAR \rightarrow id_t\ ;_t$
- $VAR \rightarrow number_t\ ;_t$
- $VAR \rightarrow string_t\ ;_t$
- $VAR \rightarrow ;_t$

Pro návrat z funkce je použité klíčové slovo *return*, kde se ověří jestli je za klíčovým slovem platná hodnota nebo jestli funkce nemá návratovou hodnotu.

**Podmínky**

- $S_{condition} \rightarrow keyword\_if_t\ S_{expression}\ \{_t\ BODY$
- $S_{condition} \rightarrow keyword\_while_t\ S_{expression}\ \{_t\ BODY$
- $BODY \rightarrow S_{condition}\ BODY$
- $BODY \rightarrow S_{statement}\ BODY$
- $BODY \rightarrow \}_t$

V podmínkách nebo cyklech se ověří jestli se náchází správné klíčové slovo, a následně se přejde do gramatiky, která ověřuje výrazy, jestli je zadaný výraz v podmínce zapsán správně. Poté nastane kontrola těla, kde se mohou nacházet příkazy a možné další podmínky na kterých je vidět rekurzivní vlastnost gramatiky.

**Podmínky ve funkcích**

- $S_{function\ condition} \rightarrow keyword\_if_t\ S_{expression}\ \{_t\ BODY$
- $S_{function\ condition} \rightarrow keyword\_while_t\ S_{expression}\ \{_t\ BODY$
- $BODY \rightarrow S_{function\ condition}\ BODY$
- $BODY \rightarrow S_{statement}\ BODY$
- $BODY \rightarrow S_{return}\ BODY$
- $BODY \rightarrow \}_t$

Podmínky ve funkcích jsou speciálním případem gramatiky normálních podmínek, kdy se ve funkcích a tudíž i v jejích podmínkách může nacházet klíčové slovo *return*, a nelze použít gramatiku pro normální podmínky pro podmínky ve funkcích. Implementačně by bylo možné tuto situaci ošetřit speciálním případem při přechodu do gramatiky podmínek a obě gramatiky sloučit do jednoho modulu, ale v rámci správného zapsání gramatik jsou použity moduly dva.

**Příkazy**

- $S_{statement} \rightarrow id_t \ =_t \ S_{assign} \ ;_t$
- $S_{statement} \rightarrow indentifier\_function_t \ S_{call} \ ;_t$
- $S_{statement} \rightarrow indentifier\_class_t \ ._t \ CLASS \ ;_t$
- $CLASS \rightarrow id_t \ =_t \ S_{assign}$
- $CLASS \rightarrow indentifier\_function_t \ S_{call}$

Mezi příkazy navrženého programovacího jazyka se řadí přiřazení do proměné nebo atributu třídy. A dále volání funkce nebo volání funkce třídy. Po kontrole co je příkaz za akci se patřičně zkontroluje pravá strana přiřazení nebo parametry volání.

**Přiřazení**

- $S_{assign} \rightarrow indentifier\_function_t \ S_{call}$
- $S_{assign} \rightarrow indentifier\_class_t \ ._t \ CLASS$
- $S_{assign} \rightarrow S_{expression}$
- $CLASS \rightarrow id_t$
- $CLASS \rightarrow indentifier\_function_t \ S_{call}$

Na pravé straně přiřazení ze může nacházet výsledek volání funkce či funkce třídy, přiřazení atributu třídy nebo matematický a logický výraz.

**Volání**

- $S_{call} \rightarrow (_t \ PARAMS$
- $PARAMS \rightarrow id_t \ PARAM$
- $PARAMS \rightarrow number_t \ PARAM$
- $PARAMS \rightarrow string_t \ PARAM$
- $PARAMS \rightarrow indentifier\_class_t \ ._t \ id_t \ PARAM$
- $PARAMS \rightarrow )_t$
- $PARAM \rightarrow ,_t \ PARAMALT \ PARAM$
- $PARAM \rightarrow )_t$
- $PARAMALT \rightarrow id_t$
- $PARAMALT \rightarrow number_t$
- $PARAMALT \rightarrow string_t$
- $PARAMALT \rightarrow identifier\_class_t \ ._t \ id_t$

Gramatika volání kontroluje správně uzávorkovaní volání a správně oddělené parametry funkce. Při volání funkce lze jako parametry použít proměnou, atribut funkce, číslo nebo řetězec.

**Výrazy**

Výrazy jsou zpracovávány precedenčním analyzátorem, místo gramatiky se zde hodí uvést použitou precedenční tabulku 6.1.

| $S_{Expression}$ | ( | ) | + | * | - | / | > | < | <= | >= | == | != | t | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | < | = | < | < | < | < | < | < | < | < | < | < | < | |
| ) | | > | > | > | > | > | > | > | > | > | > | > | | > |
| + | < | > | > | < | > | < | > | > | > | > | > | > | < | > |
| * | < | > | > | > | > | > | > | > | > | > | > | > | < | > |
| - | < | > | > | < | > | < | > | > | > | > | > | > | < | > |
| / | < | > | > | > | > | > | > | > | > | > | > | > | < | > |
| > | < | > | < | < | < | < | | | | | | | < | > |
| < | < | > | < | < | < | < | | | | | | | < | > |
| <= | < | > | < | < | < | < | | | | | | | < | > |
| >= | < | > | < | < | < | < | | | | | | | < | > |
| == | < | > | < | < | < | < | | | | | | | < | > |
| != | < | > | < | < | < | < | | | | | | | < | > |
| t | | > | > | > | > | > | > | > | > | > | > | > | | > |
| $ | < | | < | < | < | < | < | < | < | < | < | < | < | |

Tabulka 6.1: Precedenční tabulka pro uzávorkované výrazy

## 6.3 Lexikální analýza

Lexikální analýza je implementována v souborech *lex.h* a *lex.cpp* a provádí převod zdrojového jazyka na tokeny pro syntaktickou analýzu. Metoda *next_token*(), je volaná syntakctickou analýzou pro prečtení aktuálního tokenu na vstupu. Metoda je implementovaná částečně formou konečného automatu, kdy čte znaky na vstupu a pokud se znak nebo souvislost znaků, podle které se konečný automat rozhoduje, rovná některému z tokenů, vrátí daný token. Jinak vrací token, který je označován jako neplatný, což povede k neúspěchu syntaktické analýzy. Další metoda je *peek_token*(), která vrací aktuální token na vstupu, ale neprovede se jeho přečtení. Tato metoda je použita, když se gramatika rozhoduje nad vnořením, ale nechce přečíst token, který bude použit ke kontrole vnořené gramatiky.

Jelikož program neobsahuje sémantickou analýzu, je každý typ identifikátoru jiný token, s čím počítají pravidla syntaktické analýzy. Syntaktická analýza analyzuje správnost struktur, které by byly kontrolovány až sémantickou analýzou. Například při volaní funkce syntaktická analýza ví, že daný identifikátor patří právě funkci. Může podle typu identifikátoru kontrolovat, zda se jedná o volání funkce a ne přiřazení. Aby mohla lexikální analýza tyto identifikátory rozpoznat existují následující pravidla pro zápis identifikátorů. Identifikátory tříd začínají znakem podržítko (*_ClassA*). Identifikátory funkcí začínají velkým písmenem (*FunctionA*). A identifikátory proměných začínají malým písmenem (*variableA*).

V následující tabulce 6.2 je seznam tokenů a jejich význam:

| | | | |
|---|---|---|---|
| T_T | neplatný token | T_L_BRACKET | znak ( |
| T_ADD | znak + | T_R_BRACKET | znak ) |
| T_SUB | znak - | T_L_CRACKET | znak { |
| T_MUL | znak * | T_R_CRACKET | znak } |
| T_DIV | znak / | T_CLASS | identifikátor třídy |
| T_EQ | znaky == | T_FUNCTION | identifikátor funkce |
| T_NEQ | znaky != | T_IDENTIFIER | identifikátor proměné |
| T_LT | znak < | T_NUMBER | celé číslo |
| T_LTE | znaky <= | T_STRING | "řetězec" |
| T_GT | znak > | T_KEYWORD_CLASS | klíčové slovo "class" |
| T_GTE | znaky >= | T_KEYWORD_FUNCTION | klíčové slovo "function" |
| T_DOT | znak . | T_KEYWORD_RETURN | klíčové slovo "return" |
| T_SEMICOLON | znak ; | T_KEYWORD_IF | klíčové slovo "if" |
| T_COLON | znak , | T_KEYWORD_WHILE | klíčové slovo "while" |
| T_ASSIGN | znak = | T_EOF | konec vstupu |

Tabulka 6.2: Tabulka tokenů

## 6.4 Syntaktická analýza

Definice všech tříd modulů jsou obsaženy v hlavičkovém souboru *syntax.h* a definice tříd a implementace každého modulu je ve svém vlastním souboru *syntax_* ∗ *.cpp* podle dané gramatiky. Většina modulů byla implementována rekurzivním sestupem podle navržené gramatiky. Výhodou modulární implementace je, že pokud je potřeba gramatiku rozšířit nebo upravit, tak i za použití rekurzivního sestupu jsou úpravy lehce implementovatelné, bez vlivu na ostatní části systému. Jelikož každá gramatika je sama o sobě jednoduchá, tak jejich samostatná implementace není velký problém. Jediné co zbýva je ale gramatiky správně propojit a nechat celý systém pracovat dohromady.

Precedenční syntaktická analýza je použita pro analýzu matematických a logických výrazů, a je obsažena v *syntax_expression.cpp*. Syntaktická analýza pracuje podle popsaného algoritmu 2. Kromě tokenů z lexikální analýzy používá také speciální tokeny pro práci se zásobníkem, uvedeny v tabulce 6.3.

| | |
|---|---|
| T_T | dno zásobníku |
| T_TERM | terminální symbol (číslo, řetězec, proměná) |
| T_NONTERM | neterminální symbol pro derivace |
| T_OPENER | znak < na zásobníku |

Tabulka 6.3: Speciální tokeny precedenční syntaktické analýzy

## 6.5 Vstup a výstup

Program je spouštěn s jedním argumentem, název zdrojového souboru, který se má analyzovat. Následně je uveden příklad jednoduchého programu v navrženém programovacím jazyce, který lze použít jako vstup.

```
class _Person
{
    name = "test";
    age = 20 ;
    function ShowName()
    {
        Print(name);
        return name;
    }
}
x = _Person.age;
if ((x + 1) > 1)
{
    person = _Person.ShowName();
}
```

Po úspěšné analýze jazyka je na výstup strukturovaně vypsáno jak se gramatiky vnořovaly a vynořovali, je vidět postup střídaní běhu jednotlivých gramatik mezi sebou. Pokud je analýza neúspěšná je vypsán dosavadní stav gramatik a patřičná chybová hláška. Pro uvedený příklad jazyka je výstup následující.

```
Program Body
|   Class
|   |   Function
|   |   |   Statement
|   |   |   |   Calling parameters
|   |   |   |   <---
|   |   |   <---
|   |   |   Return
|   |   |   <---
|   |   <---
|   <---
|   Statement
|   |   Assigment
|   |   <---
|   <---
|   Condition
|   |   Expression
|   |   <---
|   |   Statement
|   |   |   Assigment
|   |   |   |   Calling parameters
|   |   |   |   <---
|   |   |   <---
|   |   <---
|   <---
<---
```

# Kapitola 7

# Závěr

Cílem této práce bylo uvést existující gramatické systémy a následně podle nich navrhnout vlastní gramatický systém. K navrženému systému také byla přidělena syntaktická analýza na něm založená.

Byly uvedeny dva hlavní typy gramatických systémů. Kooperačně distribuované gramatické systémy, které pracují sekvenčně a jednotlivé komponenty se střídají na tvorbě věty. Byly představeny možné derivační módy a bylo ukázáno jak tyto derivační módy mohou ovlivnit sílu gramatických systémů. Následně byly uvedeny paralelně komunikující gramatické systémy, kde každá komponenta pracuje samostaně a dochází výměně větné formy na základě komunikačního kroku. Na příkladu bylo ukázáno jak tyto gramatické systémy fungují.

Byl navrhnut nový typ gramatických systémů, inspirován částmi z obou předchozích uvedených gramatických systémů. Byť bylo předvedeno, že navržený gramatický systém se neliší od klasických gramatik, jeho strukturovaný pohled na gramatiky je užitečný v syntaktické analýze. Z tohoto pohledu byli představeny vlastnosti gramatického systému, jako je rekurze a možnost grafové reprezentace. Jako další pokračování práce by bylo možné dále studovat tyto gramatické systémy, zda se nedají objevit další zajímavé vlastnosti či rozšířit jejich síla. Jednou z možných úprav navržených gramatických systémů by mohla být místo jejich sekvenčního předávaní řízení, zdali mohou pracovat také paralelně.

Dále byla představena problematika konstrukce kompilátoru, se soustředěním na syntaktickou analýzu. Byla předvedena syntaktická analýzu shora dolů a její metoda rekurzivního sestupu. Stejně tak syntaktická analýza zdola nahoru a její metoda precedenční syntaktické analýzy. Za základě těchto metod byla popsána syntaktická analýza pro navržené gramatické systémy.

V implementační části byl vytvořen vlastní programovací jazyk, rozdělen do deseti samostatných gramatik, každá implementována ve svém modulu. Implementace ukazuje jak si mezi sebou navržené gramatické systémy předávají řízení. Implementací je tedy ověřena funkcionalita navržených gramatických systémů.

# Literatura


[1] *Chomského hierarchie* [online]. Wikipedia, The Free Encyclopedia, 2006 [cit. 2021-10-05]. Dostupné z: https://commons.wikimedia.org/wiki/File:Chomskeho_hierarchie.svg.

[2] Alfred, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. *Compilers, Principles, Techniques, and Tools, 2nd Edition*. Addison Wesley, 2007. ISBN 0-321-48681-1.

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

[4] Rozenberg, G. a Salomaa, A. *Hadbook of Formal Languages: word, language, grammar*. Springer-Verlag, Berlin, 1997. ISBN 978-3-642-63863-3.

[5] Techet, J., Masopust, T. a Meduna, A. *Cooperating Distributed Grammar Systems* [online]. 2007 [cit. 2021-10-05]. Dostupné z: https://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:09-cdgspres.pdf.

[6] Techet, J., Masopust, T. a Meduna, A. *Parallel Communicating Grammar Systems* [online]. 2007 [cit. 2021-10-05]. Dostupné z: https://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:10-pcgspres.pdf.