
**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 NĚKOLIKA GRAMATIKÁCH

**PARSING BASED ON SEVERAL GRAMMARS**

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


**AUTOR PRÁCE** **ONDŘEJ KOUMAR**
**AUTHOR**

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



**BRNO 2024**

# Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.



# Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.



# Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.



# Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.



# Citace

KOUMAR, Ondřej. *Syntaktická analýza založená na několika gramatikách.* Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexandr Meduna, CSc.

# Syntaktická analýza založená na několika gramatikách

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Ondřej Koumar
19. dubna 2024

## Poděkování

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

# Obsah



# Seznam obrázků

# Dotazy



# Todos

# Kapitola 1

# Úvod

# Kapitola 2

# Základy teorie formálních jazyků

Základní přehled matematických znalostí potřebných k pochopení této kapitoly (a tím i celé této práce) je popsán například v [13]. **[[Tento dotaz je vlastně na všechny kapitoly - snažil jsem se občas něco vymyslet, občas něco opsat z literatury, ale pořád vůbec nemám tušení, jak kapitoly uvádět. Přijde mi to takové mdlé.]]**

## 2.1 Abeceda, řetězec a jazyk

Definice v této kapitole založeny na definicích z [2, 15, 19].

### Abecedy, řetězce a operace s nimi

**Definice 2.1.1.** *Abeceda* je neprázdná množina prvků, které se nazývají *symboly* nebo *písmena*.

**Konvence 2.1.1.** Abeceda se označuje velkými písmeny řecké abecedy; v této práci bude použito výhradně písmeno $\Sigma$ (*sigma*).

**Definice 2.1.2.** Nechť $\Sigma$ je abeceda.
*Slovo* nebo *řetězec* nad abecedou $\Sigma$ je konečná posloupnost symbolů z $\Sigma$, symbol se může v řetězci několikrát opakovat. Formálně se dá zapsat jako posloupnost

$$x = (a_1, a_2, \ldots, a_n),\ n \geq 0,\ a_i \in \Sigma \text{ pro všechna } i \in \{1, \ldots, n\}\ [1].$$

Speciálním případem je *prázdný řetězec* pro $n = 0$, označuje se písmenem $\varepsilon$.

**Konvence 2.1.2.** Při zápisu řetězců je jednodušší vynechat závorky a oddělovače symbolů. Proto zápis $(a_1, a_2, \ldots, a_n)$ je ekvivalentní se zápisem $a_1 a_2 \ldots a_n$ [7].

Řetězce v této práci budou označovány malými písmeny abecedy stejně jako symboly. Pro řetězce budou používána písmena z konce abecedy ($w$, $x$, $y$, $z$ a podobně), pro symboly písmena ze začátku abecedy ($a$, $b$, $c$ a podobně).

**Definice 2.1.3.** Nechť $\Sigma$ je abeceda $x = a_1 a_2 \ldots a_n$ je řetězec nad $\Sigma$.
*Délka řetězce* $x$ je počet pozic se symboly z abecedy $\Sigma$ v řetězci, značí se $|x|$. Dále $alph(x)$ označuje množinu symbolů, které se v řetězci $x$ vyskytují.

**Definice 2.1.4.** Nechť $\Sigma$ je abeceda $x = a_1 a_2 \ldots a_n$ je řetězec nad $\Sigma$.
*Reverzace* řetězce $x$, $reversal(x)$ nebo také $x^R$ je

$$x^R = x_n x_{n-1} \ldots x_1 x_0.$$

Prázdný řetězec zřejmě bude stejný, jako jeho reverzace.

$$\varepsilon^R = \varepsilon$$

**Definice 2.1.5.** Nechť $\Sigma$ je abeceda a $w$ je řetězec nad $\Sigma$.
Řetězec $z$ je *podřetězec* řetězce $w$, jestliže existují řetězce $x$ a $y$ takové, že

$$w = xzy.$$

Řetězec $x_1$ je prefixem (předponou) řetězce $w$, jestliže existuje řetězec $y_1$ takový, že

$$w = x_1 y_1.$$

Řetězec $x_2$ je sufixem (příponou) řetězce $w$, jestliže existuje řetězec $y_2$ takový, že

$$w = y_2 x_2.$$

Je-li $y_1 \neq \varepsilon$, pak $x_1$ je *vlastní prefix* řetězce $w$; je-li $y_2 \neq \varepsilon$, pak $x_2$ je *vlastní sufix* řetězce $w$.

**Definice 2.1.6.** Nechť $\Sigma$ je abeceda.
Množina všech řetězců abecedy $\Sigma$ určité délky $k$ je $\Sigma^k$. $\Sigma^0 = \{\varepsilon\}$ pro libovolnou abecedu.

**Příklad 2.1.1.** Nechť $\Sigma = \{0, 1\}$ je abeceda.
Všechny řetězce délky 2 jsou 00, 01, 10, 11. Zřejmě tedy $\Sigma^2 = \{00,\ 01,\ 10,\ 11\}$.

**Definice 2.1.7.** Nechť $\Sigma$ je abeceda.
Množina všech řetězců nad abecedou $\Sigma$ je

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

a množina všech *neprázdných* řetězců nad abecedou $\Sigma$ je

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots \text{ nebo } \Sigma^+ = \Sigma^* \setminus \{\varepsilon\}.$$

**Definice 2.1.8.** Nechť $\Sigma$ je abeceda a $x = a_1 a_2 \ldots a_i$ a $y = b_1 b_2 \ldots b_j$ jsou řetězce nad $\Sigma$.
*Konkatenace řetězců*, symbolicky zapsána $xy$, je

$$xy = a_1 a_2 \ldots a_i b_1 b_2 \ldots b_j.$$

Jinými slovy, vznikne nový řetězec spojením $x$ a $y$ hned za sebe. Konkatenace řetězce s prázdným řetězcem v libovolném pořadí nemění původní řetězec.

$$\varepsilon x = x\varepsilon = x$$

**Příklad 2.1.2.** Nechť $\Sigma = \{0, 1\}$ je abeceda, $x = 011$, $y = 1101$ řetězce nad $\Sigma$.
Konkatenace řetězců $xy = 0111101$ a $yx = 1101011$.

### Jazyky a operace s nimi

**Definice 2.1.9.** Nechť $\Sigma$ je abeceda.
Jazyk $L$ nad abecedou $\Sigma$ je konečná či nekonečná množina řetězců nad $\Sigma$, nebo také $L \subseteq \Sigma^*$.
Řetězce jazyka se nazývají *věty* nebo *slova*.

Množiny $\emptyset$ a $\{\varepsilon\}$ jsou jazyky nad libovolnou abecedou $\Sigma$, nicméně $\emptyset \neq \{\varepsilon\}$. $\emptyset$ je jazyk, který neobsahuje žádné řetězce, zatímco $\{\varepsilon\}$ obsahuje právě jeden řetězec, a to prázdný.

**Definice 2.1.10.** Nechť $L_1$ a $L_2$ jsou jazyky nad $\Sigma$.
Typické množinové operace mohou jsou použitelné i pro jazyky:

- sjednocení jazyků je definováno jako

$$L_1 \cup L_2 = \{x : x \in L_1 \vee x \in L_2\},$$

- průnik jazyků je definován obdobně jako

$$L_1 \cap L_2 = \{x : x \in L_1 \wedge x \in L_2\},$$

- rozdíl jazyků je

$$L_1 \setminus L_2 = \{x : x \in L_1 \wedge x \notin L_2\}.$$

Pro jazyky existuje speciální operace *konkatenace*, která vychází z konkatenace řetězců:

$$L_1 L_2 = \{xy : x \in L_1 \wedge y \in L_2\}.$$

**Definice 2.1.11.** Nechť $\Sigma$ je abeceda a $L$ jazyk nad $\Sigma$.
*Reverzace* jazyka, $reversal(L)$ nebo $L^R$ je

$$L^R = \{x^R : x \in L\},$$

*mocnina* jazyka $L^i$ je definována jako

$$\begin{aligned} &1) \quad L^0 = \{\varepsilon\}, \\ &2) \quad L^i = LL^{i-1},\ i \geq 0, \end{aligned}$$

*iterace* jazyka je sjednocení všech mocnin jazyka:

$$L^* = \bigcup_{n \geq 0} L^n,$$

*pozitivní iterací* rozumíme obyčejnou iteraci bez nulté mocniny:

$$L^+ = \bigcup_{n \geq 1} L^n.$$

Z těchto definic dále vyplývá

$$\begin{aligned} L^* &= L^+ \cup \{\varepsilon\}, \\ L^+ &= L^* L = L L^*, \end{aligned}$$

důkaz lze nalézt například v [19].

## 2.2 Gramatika

Informace pro tuto kapitolu byly převzaty z [19].

V kapitole 2.1 byly ukázány obecné a pouze základní principy jazyků. S triviálními způsoby reprezentace jazyka, tedy například výčtem všech vět, si nevystačíme už jen díky tomu, že jazyky často bývají nekonečné. Gramatiky jsou jedním ze způsobů, a to velmi elegantním, jak jazyky reprezentovat.

Používají dva typy symbolů – *nonterminální* symboly a *terminální* symboly. Nonterminální symboly slouží k popisu syntaktických celků jazyka a terminální symboly se shodují se symboly ze vstupní abecedy.

**Konvence 2.2.1.** Pro nonterminální symboly bude dále v práci používáno zkrácené označení *neterminály* a pro terminální symboly pojem *terminály*.

**Konvence 2.2.2.** V kontextu práce s gramatikami bude nadále používáno následující označení pro různé typy řetězců:

- malá písmena ze začátku abecedy ($a$, $b$, $c, \ldots$) budou reprezentovat terminály,
- velká písmena ze začátku abecedy ($A$, $B$, $C, \ldots$) budou vyhrazena pro neterminály,
  - výjimkou je symbol $S$, který je často používán pro označení *startovacího symbolu*,
- malá písmena ze začátku řecké abecedy ($\alpha$, $\beta$, $\gamma, \ldots$) budou označovat řetězce terminálů a neterminálů,
- malá písmena z konce abecedy ($u$, $v$, $w, \ldots$) budou pro řetězce terminálů.

Gramatika představuje generativní systém, ve kterém lze z jistého vyznačeného neterminálu generovat, aplikací *přepisovacích pravidel*, řetězce tvořené neterminálními a terminálními symboly, které nazýváme *větnými formami*. Větné formy, které jsou tvořeny pouze terminálními symboly, reprezentují věty gramatikou definovaného jazyka. Pokud gramatika negeneruje žádnou větu, pak reprezentuje prázdný jazyk.

Přepisovací pravidla jsou jádrem gramatik. Jejich využití spočívá v neustálém přepisování neterminálních symbolů, dokud není vygenerován řetězec terminálů (podrobněji v definicích 2.2.2, 2.2.3 a 2.2.4). Každé pravidlo je ve tvaru uspořádané dvojice $(\alpha, \beta)$ řetězců, kdy se při aplikaci tohoto pravidla podřetězec $\alpha$ nahradí řetězcem $\beta$. Podmínkou je, aby $\alpha$ obsahoval alespoň jeden neterminál.

**Definice 2.2.1.** Gramatika $G$ je čtveřice $G = (N, T, P, S)$, kde

- $N$ je konečná množina neterminálů,
- $S$ je konečná množina terminálů,
- $P$ je konečná relace $R \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$,
- $S$ je počáteční (výchozí, startovací) symbol gramatiky.

Prvek $(\alpha, \beta) \in P$ (*přepisovací pravidlo*) bývá zjednodušeně zapisováno jako $\alpha \to \beta$.

V následujícím příkladě je ukázka gramatiky v obecném tvaru.

**Příklad 2.2.1.**

$$G = (\{A, S\},\ \{0, 1\},\ P,\ S)$$
$$P = \{S \to 0A1,\ 0A \to 00A1, A \to \varepsilon\}$$

### Derivační krok

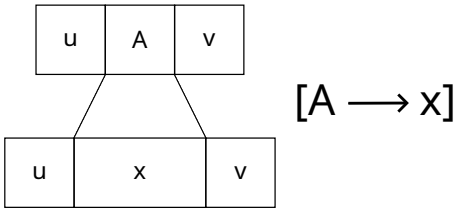
Myšlenkou derivačního kroku je přepsat aktuální řetězec na řetězec nový za použití přepisovacího pravidla $p \in P$.

**Definice 2.2.2.** Nechť $G = (N, T, P, S)$ je gramatika, $\lambda, \mu \in (N \cup T)^*$ a $p = \alpha \to \beta \in P$. Mezi řetězci $\lambda$ a $\mu$ platí binární relace $\underset{G}{\Rightarrow}$, nazývaná *přímá derivace*, jestliže můžeme řetězce $\lambda$ a $\mu$ vyjádřit ve tvaru

$$\lambda = \gamma\alpha\delta$$
$$\mu = \gamma\beta\delta$$

kde $\gamma$ a $\delta$ jsou libovolné řetězce z $(N \cup T)^*$. Říkáme také, že gramatika $G$ provádí *derivační krok* z $\gamma\alpha\delta$ do $\gamma\beta\delta$, respektive z $\lambda$ do $\mu$.

To znamená, že pokud lze přepsat řetězec $\lambda = \gamma\alpha\delta$ na řetězec $\mu = \gamma\beta\delta$ za použití pravidla $\alpha \to \beta$, potom mezi řetězci $\alpha$ a $\beta$ je relace přímé derivace, zapsáno $\alpha \Rightarrow \beta$.




Obrázek 2.1: Ilustrace derivačního kroku za použití pravidla $\alpha \to \beta$.
[[**zmenit pismenka**]]

### Sekvence derivačních kroků

**Definice 2.2.3.** Nechť $G = (N, T, P, S)$ je gramatika a $\lambda, \mu \in (N \cup T)^*$.
Mezi řetězci $\lambda$ a $\mu$ platí relace $\Rightarrow^+$ nazývaná *derivace*, jestliže existuje posloupnost přímých derivací $\chi_{i-1} \Rightarrow \chi_i, i \in \{1, \ldots, n\}, n \geq 1$ taková, že platí

$$\lambda = \chi_0 \Rightarrow \chi_1 \Rightarrow \ldots \Rightarrow \chi_{n-1} \Rightarrow \chi_n = \mu.$$

Tato posloupnost se nazývá *derivace délky n*. Platí-li $\lambda \Rightarrow \mu$, pak řetězec $\mu$ lze *generovat* z řetězce $\lambda$, nebo také $\mu$ je *derivovatelný* z $\lambda$ v gramatice $G$. Relace $\Rightarrow^+$ je tranzitivním uzávěrem relace přímé derivace $\Rightarrow$. Symbolem $\Rightarrow^n$ se značí n-tá mocnina přímé derivace $\Rightarrow$.

Jinými slovy, pokud řetězec $\lambda$ derivuje řetězec $\chi$ v nenulovém počtu kroků a zároveň $\chi$ derivuje řetězec $\mu$ v nenulovém počtu kroků, pak je zřejmé, že $\lambda$ derivuje $\mu$ v nenulovém počtu kroků, zapsáno $\lambda \Rightarrow^+ \mu$.

**Definice 2.2.4.** Nechť $G = (N, T, P, S)$ je gramatika a $\lambda, \mu \in (N \cup T)^*$.
Jestliže v $G$ platí pro řetězce $\lambda$ a $\mu$ relace $x \Rightarrow^+ y$ nebo identita $\lambda = \mu$, pak $\lambda \Rightarrow^* \mu$. Relace $\Rightarrow^*$ je reflexivním a tranzitivním uzávěrem relace přímé derivace $\Rightarrow$.

Reflexivní uzávěr relace přímé derivace $\Rightarrow$ znamená, že řetězec přímo derivuje sám sebe v nula krocích. Také je možné říci, že se nepoužije žádné pravidlo k přepsání řetězce na sebe sama, zapsáno $\lambda \Rightarrow^0 \lambda$.

## 2.3 Chomského hierarchie gramatik

Informace k následující kapitole převzaty z [19].

Gramatiky se dělí do čtyř skupin podle tvaru přepisovacích pravidel. Označují se jako typ 0, typ 1, typ 2, typ 3, respektive neomezené gramatiky, kontextové gramatiky, bezkontextové gramatiky a pravé lineární gramatiky. Tyto gramatiky generují příslušné jazyky $L_i$, $i \in \{0, 1, 2, 3\}$, $i$ je příslušné s typem gramatiky. Platí $L_0 \subseteq L_1 \subseteq L_2 \subseteq L_3$.

### Neomezená gramatika

Gramatika typu 0 obsahuje pravidla v nejobecnějším tvaru, shodným z definice 2.2.1.

$$\alpha \to \beta,\ \alpha \in (N \cup T)^* N (N \cup T)^*,\ \beta \in (N \cup T)^*$$

### Kontextová gramatika

Gramatika typu 1 obsahuje pravidla ve tvaru:

$$\alpha A \beta \to \alpha \gamma \beta,\ A \in N,\ \alpha, \beta \in (N \cup T)^*,\ \gamma \in (N \cup T)^+$$

nebo $S \to \varepsilon$, pokud se $S$ nevyskytuje na pravé straně žádného pravidla.

**Příklad 2.3.1.** Příklad kontextové gramatiky:

$$\begin{aligned} G &= (\{A, S\}, \{0, 1\}, P, S) \text{ s pravidly} \\ S &\to 0A \mid \varepsilon \\ 0A &\to 00A1 \\ A &\to 1 \end{aligned}$$

Těmto gramatikám se říká *kontextové*, protože neterminál $A$ může být přepsán řetězcem $\gamma$ pouze tehdy, když jeho levým kontextem je řetězec $\alpha$ a jeho pravým kontextem řetězec $\beta$. Tyto gramatiky nepřipouštějí, aby neterminál byl nahrazen prázdným řetězcem, tedy zakazuje pravidla ve tvaru

$$\alpha A \beta \to \alpha \beta,$$

kromě výše zmiňované výjimky se startovacím symbolem $S$.

### Bezkontextová gramatika

Práce s gramatikami typu 2 je jedním z jader této práce. Definujme tedy tento typ gramatiky podrobněji než ostatní typy.

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

- $N$, $T$, $S$ jsou definovány stejně jako v definici 2.2.1,
- $P$ je množina přepisovacích pravidel ve tvaru $A \to \gamma$, $A \in N$ a $\gamma \in (N \cup T)^*$,
    - je tudíž podmnožinou kartézského součinu $P \subseteq N \times (N \cup T)^*$.

**Konvence 2.3.1.** Pro označení bezkontextových gramatik bude v textu dále využívána zkratka BKG.

Substituci neterminálu $A$ je možné provést bez závislosti na pravém a levém kontextu, ve kterém je neterminál $A$ uložen. Tyto gramatiky smějí obsahovat pravidla ve tvaru $A \to \varepsilon$.

**Příklad 2.3.2.** Příklad bezkontextové gramatiky:

$$\begin{aligned} G &= (\{S\}, \{0, 1\}, P, S) \text{ s pravidlem} \\ S &\to 0S1 \mid \varepsilon \end{aligned}$$

### Pravá lineární gramatika

Gramatika typu 3 obsahuje pravidla ve tvaru

$$A \to xB \text{ nebo } A \to x,\ A, B \in N,\ x \in T^*.$$

Jediný možný neterminál v pravidlech stojí úplně vpravo, proto *pravá lineární* gramatika. Další možný název je *regulární gramatika*.

**Příklad 2.3.3.** Příklad pravé lineární gramatiky:

$$\begin{aligned} G &= (\{A, B\}, \{a, b, c\}, P, S) \text{ s pravidly} \\ A &\to aaB \mid ccB \\ B &\to bB \mid \varepsilon \end{aligned}$$

## 2.4 Konečný automat

Definice související s konečnými automaty převzaty z [5], není-li řečeno jinak.

**Definice 2.4.1.** *Konečný automat* je pětice

$$M = (Q, \Sigma, R, s, F),$$

kde

- $Q$ je konečná množina stavů,
- $\Sigma$ je konečná vstupní abeceda, $Q \cap \Sigma = \emptyset$,
- $R$ je konečná relace, $R \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$,
  - je nazývána *množinou pravidel* ve tvaru $qa \to p,\ q, p \in Q,\ a \in \Sigma \cup \{\varepsilon\}$. Jakákoli uspořádaná trojice $(q, a, p) \in R$ je pravidlem, zapsáno $qa \to p$.
  - Další možnost zápisu je zobrazení (*přechodová funkce*) $R : Q \times \Sigma \to 2^Q$ [19].
- $s \in Q$ je počáteční stav automatu,
- $F \subseteq Q$ je konečná množina koncových stavů.

**Konvence 2.4.1.** Pro označení konečných automatů bude dále v textu použito zkrácené KA.

U konečných automatů popisujeme jejich *konfiguraci* – ve kterém stavu se nachází a jaký řetězec na vstupní pásce.

**Definice 2.4.2.** Nechť $M = (Q, \Sigma, R, s, F)$ je KA. Dle autorů v [19] je konfigurace $M$ uspořádaná dvojice

$$(q, w) \in Q \times \Sigma^*.$$

Nechť $X_M$ značí množinu všech konfigurací $M$.

Pomocí konfigurací můžeme definovat *přechody* KA, které reprezentují jejich výpočetní kroky. Výpočetní krok znamená přechod z jedné konfigurace do druhé za přečtení symbolu ze vstupní pásky.

**Definice 2.4.3.** Nechť $M = (Q, \Sigma, R, s, F)$ je KA.
Binární relace $\vdash_M$, nazývána přechodem KA $M$, je

$$\beta \vdash_M \chi,$$

kde $\beta = (q, ax)$, $\chi = (p, x) \in X_M$, a $qa \to p \in R$. Je to binární relace na množině konfigurací – pokud $\beta \vdash_M \chi$, pak $(\beta, \chi) \in X_M \times X_M$. Pokud $a = \varepsilon$, není ze vstupní pásky přečten žádný symbol.

Symboly $\vdash_M^n, \vdash_M^+$ a $\vdash_M^*$ nechť značí příslušně $n$-tou mocninu pro $n \geq 0$, tranzitivní a reflexivně-tranzitivní uzávěr relace $\vdash_M$ podobně jako u derivačního kroku gramatik (2.2.2), čímž reprezentují *sekvenci přechodů* KA $M$. Například pro konfigurace $\beta$ a $\chi$ platí $\beta \vdash_M^+ \chi$ právě tehdy, když

$$\beta = c_0 \vdash_M c_1 \vdash_M \ldots \vdash_M c_{n-1} \vdash_M c_n = \chi,$$

pokud $\beta, \chi, c_1, \ldots, c_{n-1} \in X_M$, $n \geq 1$.

**Konvence 2.4.2.** Bude-li z kontextu jasné, že se jedná o přechod automatu $M$, pak bude relace přechodu $\vdash$ psána bez indexu $M$.

**Definice 2.4.4.** Nechť $M = (Q, \Sigma, R, s, F)$ je KA.
Jazyk přijímaný $M$, značen $L(M)$, je

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

Jsou to řetězce takové, po jejichž zpracování skončí $M$ v koncovém stavu.

**Definice 2.4.5.** Nechť $M = (Q, \Sigma, R, s, F)$ je KA.
$M$ je *KA bez $\varepsilon$-přechodů*, pokud pro každé pravidlo $qa \to p \in R$ platí, že $a \neq \varepsilon$.

**Definice 2.4.6.** Nechť $M = (Q, \Sigma, R, s, F)$ je KA bez $\varepsilon$-přechodů.
$M$ je *deterministický KA* právě tehdy, když pro každé $q \in Q$ a každé $a \in \Sigma$ neexistuje více než jedno $p \in Q$ takové, že $qa \to p \in R$. Jinými slovy, z jednoho stavu není možné přejít do několika stavů přečtením stejného symbolu.

**[[jendoduchy priklad KA i s prechodovym diagramem]]**

## 2.5 Zásobníkový automat

Zásobníkový automat je rozšíření konečného automatu, popsaného v definici 2.4.1, o zásobník. Zásobník slouží jako paměťové médium, na které si automat může ukládat informace v podobě symbolů zásobníkové abecedy. Následující definice převzaty z [5, 19].

**Definice 2.5.1.** *Zásobníkový automat* (ZA) je sedmice

$$M = (Q, \Sigma, \Gamma, R, s, S, F),$$

kde:

- $Q, \Sigma, s, F$ jsou definovány stejně jako v definici 2.4.1,
- $\Gamma$ je konečná zásobníková abeceda,
- $R$ je konečná relace $R \subseteq \Gamma \times Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \times Q$ nazývána množinou pravidel ZA,

- každé pravidlo $(A, q, a, w, p)$ může být zapsáno ve tvaru $Aqa \to wp$, kde $A \in \Gamma$, $q, p \in Q,\ a \in \Sigma \cup \{\varepsilon\},\ w \in \Gamma^*$ [10],

- $S \in \Gamma$ je počáteční symbol na zásobníku.

**Konvence 2.5.1.** Zásobníkové automaty budou dále značeny zkráceně ZA.

**Definice 2.5.2.** Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je ZA.
Konfigurací ZA nazveme trojici

$$(\alpha, q, w) \in \Gamma^* \times Q \times \Sigma^*,$$

kde

- $\alpha$ je obsah zásobníku,
- $q$ je aktuální stav automatu (*řídící jednotky*),
- $w$ je doposud nepřečtená část vstupního řetězce, jehož první symbol je aktuálně pod čtecí hlavou. Pokud $w = \varepsilon$, všechny symboly byly ze vstupní pásky již přečteny.

Nechť $X_M$ reprezentuje množinu všech konfigurací ZA $M$.

**Definice 2.5.3.** Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je ZA.
Přechod ZA je binární relace definována nad množinou konfigurací (podobně jako v definici 2.4.3) v podobě

$$\beta \vdash \chi,$$

kde $\beta = (uA, q, av),\ \chi = (uw, p, v)$ a $Aqa \to wp \in R$. Symbolem $A$ je reprezentován vrchol zásobníku a symbol $a$ nechť reprezentuje aktuální symbol pod čtecí hlavou. Pokud $w = \varepsilon$, pak se pouze vyjme $A$ ze zásobníku bez náhrady. Relace $\vdash^n$, $\vdash^+$, $\vdash^*$ jsou opět $n$-tou mocninou relace, tranzitivním a reflexivně-tranzitivním uzávěrem.

Oproti klasickým KA se při přechodu musí pracovat se zásobníkem, ze kterého se vyjme symbol $A$, namísto něj se vloží symbol $w$.

Automat $M$ přijímá řetězec právě tehdy, když $Ssw \vdash^* uf$ v $M$; $u \in \Gamma^*, f \in F$.

**Definice 2.5.4.** Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je ZA.
Jazyk přijímaný $M$ je

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

**Příklad 2.5.1.** Nechť $M = (\{s, f\}, \{a, b\}, \{S, a\}, R, s, S, \{f\})$ je ZA a

$$R = \{Ssa \to as,\ asas \to aas,\ asb \to f,\ afb \to f\}.$$

Počáteční konfigurace nechť je $(S, s, aaabbb)$. Přechody $M$ budou vypadat následovně:

$$\begin{aligned}
(S, s, aaabbb) &\vdash (a, s, aabbb) && [Ssa \to as] \\
&\vdash (aa, s, abbb) && [asa \to aas] \\
&\vdash (aaa, s, bbb) && [asa \to aas] \\
&\vdash (aa, f, bb) && [asb \to f] \\
&\vdash (a, f, b) && [afb \to f] \\
&\vdash (\varepsilon, f, \varepsilon) && [afb \to f]
\end{aligned}$$

**[[Jestli bude cas - pridat prechodovy diagram]]**

### Rozšířený zásobníkový automat

Rozšířené zásobníkové automaty reprezentují přirozené rozšíření klasických ZA, které na zásobníku mohou pracovat pouze s jeho vrcholem. Rozšířené zásobníkové automaty mohou provádět expanzi symbolů na zásobníku v libovolné hloubce, jinak pracují indenticky. Text a definice v této kapitole převzaty z [5, 13], není-li řečeno jinak.

**Definice 2.5.5.** *Rozšířený zásobníkový automat* (RZA) je sedmice

$$M = (Q, \Sigma, \Gamma, R, s, S, F),$$

kde:

- $Q, \Sigma, s, S, F$ jsou definovány stejně jako u klasických ZA (2.5.1),
- $\Gamma$ je zásobníková abeceda, $\mathbb{N}$, $Q$, a $\Gamma$ jsou navzájem disjunktní, $\Sigma \subseteq \Gamma$[[**opravdu $\subseteq$ a ne $\subset$?**]]

  a $\Gamma \setminus \Sigma$ obsahuje speciální symbol $\#$ (*spodní symbol*), který je považován za dno zásobníku,
- $R$ je konečná relace

$$\begin{aligned}&(\mathbb{N} \times Q \times (\Gamma \setminus (\Sigma \cup \{\#\})) \times Q \times (\Gamma \setminus \{\#\})^+) \cup \\ &(\mathbb{N} \times Q \times \{\#\} \times Q \times (\Gamma \setminus \{\#\})^*\{\#\}),\end{aligned}$$

  místo uspořádané pětice $(m, q, A, p, v) \in R$ píšeme $mqA \to pv \in R$ a $R$ nazýváme množinou pravidel.

  - Z definice $R$ lze vidět, že jsou dva typy přechodových pravidel, dají se zapsat ve tvaru

$$\begin{aligned}&mqA \to pv, \\ &mq\# \to pv\# \text{ [8]}.\end{aligned}$$

    O jejich použití a rozdílech je psáno v definici 2.5.7.

**Konvence 2.5.2.** Rozšířené zásobníkové automaty budou dále označeny zkratkou RZA.

Přechody RZA pracují, stejně jako přechody ZA, nad konfiguracemi. Konfigurace jsou podobné těm u klasických ZA, nicméně stav zásobníku musí končit symbolem $\#$ a zbytek řetězce na zásobníku jej nesmí obsahovat.

**Definice 2.5.6.** Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA.
Konfigurací RZA nazveme trojici

$$(q, w, \alpha) \in Q \times \Sigma^* \times (\Gamma \setminus \{\#\})^*\{\#\}.$$

Nechť $X_M$ reprezentuje množinu všech konfigurací RZA $M$.

Další podobnost je práce s řetězci abedecedy $\Sigma$ a s pravidly při přechodech. Jediná změna od klasických ZA již byla zmíněna na začátku této kapitoly – při přechodech se na vrcholu zásobníku mohou měnit celé řetězce.

**Definice 2.5.7.** Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA a $\beta,\ \chi \in X_M$ jeho konfigurace. $M$ vyjme (anglicky *pops*) ze zásobníku symbol a přejde z konfigurace $\beta$ do konfigurace $\chi$ za přečtení symbolu vstupní pásky, symbolicky zapsáno

$$\beta\ {}_p\vdash \chi,$$

pokud konfigurace jsou ve tvaru $\beta = (q, au, az),\ \chi = (q, u, z),\ a \in \Sigma,\ u \in \Sigma^*,\ z \in \Gamma^*$. $M$ rozšíří (anglicky *expands*) symbol na zásobníku a přejde z konfigurace $\beta$ do konfigurace $\chi$ za přečtení symbolu ze vstupní pásky, symbolicky zapsáno

$$\beta\ {}_e\vdash \chi,$$

pokud konfigurace jsou ve tvaru $\beta = (q, w, uAz),\ \chi = (p, w, uvz)$ a zároveň $mqA \to pv \in R$; $q, p \in Q,\ w \in \Sigma^*,\ A \in \Gamma,\ u, v, z \in \Gamma^*$; $m$ je hloubka symbolu $A$ v zásobníku, respektive $u$ obsahuje $m-1$ *nevstupních* symbolů (symboly $a$, pro které platí $\{a : a \in \Gamma \setminus \Sigma\}$). Pro ilustraci, že automat udělá přechod $\beta\ {}_e\vdash \chi$ podle pravidla $mqA \to pv$, píšeme

$$\beta\ {}_e\vdash \chi\ \ [mqA \to pv].$$

$M$ udělá *přechod* z $\beta$ do $\chi$, psáno

$$\beta \vdash \chi$$

právě tehdy, když $M$ udělá vyjmutí symbolu $\beta\ {}_p\vdash \chi$ nebo expanzi symbolu $\beta\ {}_e\vdash \chi$. Tranzitivní uzávěry ${}_p\vdash^+, {}_e\vdash^+$, reflexivně-tranzitivní uzávěry ${}_p\vdash^*, {}_e\vdash^*$ a $n$-té mocniny ${}_p\vdash^n, {}_e\vdash^n$ jsou definovány standardně. Alternativní definice přechodu je popsána v [19].

Obrázek 2.2: Ilustrace p-přechodu a e-přechodu RZA
**[[pridat obrazek]]**

Pokud existuje $n \in \mathbb{N}$ takové, že pro všechna pravidla $M$ platí, že jejich hloubka je maximálně rovna $n$, pak říkáme, že $M$ je hloubky maximálně $n$, zapsáno ${}_nM$.

**Definice 2.5.8.** Nechť ${}_nM = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA hloubky $n \in \mathbb{N}$.
Jazyk přijímaný automatem $M$ je

$$L({}_nM) = \{w \in \Sigma^* : (s, w, S\#) \vdash^* (f, \varepsilon, \#) \text{ v } {}_nM,\ f \in F\}.$$

Tyto jazyky jsou přijímané vyprázdněním zásobníku a zároveň přechodem do koncového stavu automatu. Dále existují jazyky přijímané RZA pouze vyprázdněním zásobníku, přičemž automat se nemusí dostat do koncového stavu.

**Definice 2.5.9.** Nechť ${}_nM = (Q, \Sigma, \Gamma, R, s, S, F)$ je RZA hloubky $n \in \mathbb{N}$.
Jazyk přijímaný automatem M vyprázdněním zásobníku je

$$E({}_nM) = \{w \in \Sigma^* : (s, w, S\#) \vdash^* (q, \varepsilon, \#) \text{ v } {}_nM,\ q \in Q\}.$$

**Příklad 2.5.2.** Nechť ${}_2M = (\{s, q, p, f\}, \{a, b, c\}, \Sigma \cup \{A, S, \#\}, R, s, S, \{f\})$, kde $R$ je

$$\begin{array}{lll} 1sS \to qAA, & 1qA \to fab, & 1fA \to fc, \\ 1qA \to paAb, & 2pA \to qAc. & \end{array}$$

Mějme počáteční konfiguraci $(s, aabbcc, S\#)$. $M$ udělá nejdříve dvakrát expanzi – jednou expanduje startovací symbol na $AA$, přičemž $A$ není vstupní symbol, takže expanze proběhne podruhé.

$$\begin{aligned}(s, aabbcc, S\#) &\ {}_e\vdash (q, aabbcc, AA\#) && [1sS \to qAA]\\ &\ {}_e\vdash (p, aabbcc, aAbA\#) && [1qA \to paAb]\end{aligned}$$

Na aktuální konfiguraci je vidět, že na vrcholu zásobníku i aktuálně čtený symbol je $a$. $M$ tento symbol vyjme.

$$(p, aabbcc, aAbA\#) \ {}_p\vdash (p, abbcc, AbA\#)$$

Následující krok bude znovu expanze, tentokrát hlouběji v zásobníku.

$$(p, abbcc, AbA\#) \ {}_e\vdash (q, abbcc, AbAc\#) \quad [2pA \to qAc]$$

Dále přijímání řetězce probíhá stejným způsobem jako v předešlých krocích. **[[Příklad v knize - přeskočení kroku z 5 na 6, z kroku 7 na 8 je psána expanze, ale nic se v konfiguraci nezměnilo - vysvětlím osobně]]**

$$\begin{aligned}(q, abbcc, AbAc\#) &\ {}_e\vdash (q, abbcc, abbAc\#) && [1qA \to fab]\\ &\ {}_p\vdash (f, bbcc, bbAc\#)\\ &\ {}_p\vdash (f, bcc, bAc\#)\\ &\ {}_p\vdash (f, cc, Ac\#)\\ &\ {}_e\vdash (f, cc, cc\#) && [1fA \to fc]\\ &\ {}_p\vdash (f, c, c\#)\\ &\ {}_p\vdash (f, \varepsilon, \#)\end{aligned}$$

# Kapitola 3

# Gramatické systémy

[[v cele kapitole refaktorovat pismenka dle konvence]]

Veškeré definice a znalosti použité v této kapitole převzaty z [16, 17, 14], není-li řečeno jinak.

## 3.1 Kooperující distribuované gramatické systémy

Kooperující distribuovaný (*cooperating distributed*) gramatický systém stupně $n$ je systém gramatik, které mezi sebou sdílejí množinu neterminálů i terminálů a startovací symbol. Spolupracují mezi sebou předáváním řízení derivace aktuálně zpracovávaného řetězce dle pravidel nastaveného *derivačního režimu*.

**Definice 3.1.1.** Kooperující distribuovaný gramatický systém je $(n+3)$-tice

$$\Gamma = (N, T, S, P_1, \ldots, P_n),$$

kde:

- $N$, $T$, a $S$ jsou definovány stejně jako v definici 2.3.1,
- $P_i$ je konečná množina pravidel ve tvaru $A \to x$, kde pravidla jsou definována stejně jako v definici 2.3.1, nazývaná *komponentou* systému, $i \in \{1, \ldots, n\}$,
- $i$-tá gramatika systému se zapisuje jako $G_i = (N, T, S, P_i)$

Alternativní definice pro CDGS je $\Gamma = ((N, T, S, P_1), \ldots, (N, T, S, P_n))$.

**Konvence 3.1.1.** Pro označení kooperujících distribuovaných gramatických systémů bude dále využito zkrácené *CDGS*, případně *CD gramatické systémy*.

### Derivační krok v CDGS

Notace derivačního kroku v CDGS je

$$x \,{}_i\Rightarrow^f y.$$

Tento zápis znamená, že řetězec $x \in (N \cup T)^*$ derivuje řetězec $y \in (N \cup T)^*$ v $i$-té komponentě za použití *derivačního režimu* $f$.

### Derivační režimy

Prvním příkladem je režim $*$, který byl v definici 2.2.4 uveden pro bezkontextovou gramatiku a princip v CD gramatických systémech je podobný. Při použití tohoto derivačního režimu řetězec $x \in (N \cup T)^*$ derivuje řetězec $y \in (N \cup T)^*$ v libovolném počtu kroků v $i$-té komponentě, zapsáno $x \,{}_i\Rightarrow^* y$. Je možné předat řízení derivace jiné komponentě i v případě, že ve stejné komponentě lze s derivací stejného řetězce pokračovat.

Podobným příkladem je režim *ukončovací*, který spočívá v nutné derivaci řetězce v dané komponentě, dokud je to možné. Značí se písmenem $t$. Jsou dvě nutné podmínky, aby $y$ bylo derivovatelné z $x$ v komponentě $G_i$ režimem $t$.

1. $x \,{}_i\Rightarrow^* y$ – v dané komponentě lze posloupností derivačních kroků získat řetězec $y$ z řetězce $x$,
2. $y \,{}_i\not\Rightarrow z$ pro všechna $z \in (N \cup T)^*$ – ve stejné komponentě nelze nalézt další pravidlo, které by derivovalo $y$.

Jinými slovy, pokud můžeme z aktuálního řetězce v aktuální komponentě odvodit řetězec nový, *musí* se s derivací v této komponentě pokračovat.

Další derivační režimy:

- alespoň $k$ derivací, tedy $x \,{}_i\Rightarrow^{\geq k} y$,
- nejvíce $k$ derivací, tedy $x \,{}_i\Rightarrow^{\leq k} y$,
- právě $k$ derivací, tedy $x \,{}_i\Rightarrow^{=k} y$,

kde $k \in \mathbb{N} \cup \{0\}$ a $i$ symbolizuje $i$-tou komponentu gramatického systému.

Derivačních režimů existuje mnohem více**[[Zdroj?? - Mluvili jsme o tom na konzultaci a říkáte to i v záznamu přednášky TID. Stačil by asi klidně jen nějaký odkaz na výčet. ]]**

, nejsou ale pro tuto práci podstatné. Tyto režimy mohou být reprezentovány jako množina, což pomůže definovat další pojmy v následující podkapitole o generovaném jazyce.

**Definice 3.1.2.** Nechť $k \in \mathbb{N}$ a $*$, $t$ představují derivační režimy.
Potom množina

$$D = \{*, t\} \cup \{\leq k, \geq k, = k\}$$

reprezentuje derivační režimy použitelné v CD gramatických systémech.

### Jazyk generovaný CD gramatickým systémem

Než bude definován samotný jazyk, je vhodné definovat pomocnou množinu, která reprezentuje *možné derivace* z řetězců.

**Definice 3.1.3.** Nechť $\Gamma = (N, T, S, P_1, \ldots, P_n)$.
Potom

$$F(G_j, u, f) = \{v : u_j \Rightarrow^f v\},\ j \in \{1, \ldots, n\},\ f \in D,\ u \in (N \cup T)^*$$

je množina všech řetězců $v$ derivovatelných z $u$ v $j$-té komponentě za použití derivačního režimu $f$.

**Definice 3.1.4.** Nechť $\Gamma = (N, T, S, P_1, \ldots, P_n)$.
Jazyk generovaný systémem $\Gamma$ za derivačního režimu $f$ je

$$L_f(\Gamma) = \{w \in T^* : \text{ existují } v_0, v_1, \ldots, v_m \text{ takové, že } v_k \in F(G_{j_k}, v_{k-1}, f), \\ k \in \{1, \ldots, m\},\ j_k \in \{1, \ldots, n\},\ v_0 = S,\ v_m = w \text{ pro } m \geq 1\}.$$

Výsledný řetězec $w$, který vznikl postupnou derivací startovacího symbolu $v_0$. Měl několik mezikroků, které jsou reprezentovány řetězci $v_1, \ldots, v_{m-1}$. Každý řetězec $v_k$, kde $k \in \{1, \ldots, m\}$ byl zderivován z řetězce $v_{k-1}$ v komponentě $G_{j_k}$, kde $j_k \in \{1, \ldots, n\}$, za derivačního režimu $f$.

**Příklad 3.1.1.** Nechť $\Gamma = (\{S, A\}, \{a\}, S, P_1, P_2, P_3)$, kde $P_1 = \{S \to AA\}$, $P_2 = \{A \to S\}$, $P_3 = \{A \to a\}$. Nechť $f = t$, $\Gamma$ pracuje na ukončovacím derivačním režimu.

Počáteční řetězec je počáteční symbol, tedy $S$. Je pouze jedna možnost, jak $S$ přepsat, a to použitím pravidla z $P_1$, $S \to AA$.

$$S \to AA \quad [S \to AA]$$

Díky tomu, že v komponentě $P_1$ už neexistuje pravidlo, kterým by mohla derivace dále pokračovat, řízení derivace je předáno komponentě jiné. Aktuálně zpracovávaný řetězec je $AA$. Komponenty $P_1$ a $P_2$ mají pravidla pro přepsání neterminálu $A$. Při použití derivačního režimu $t$ je zřejmé, že celý řetězec bude přepisovat pouze jedna komponenta. Při předání komponentě $P_3$ je řetězec přepsán na $aa$,

$$AA \to aA \quad [A \to a]$$
$$aA \to aa \quad [A \to a]$$

komponenta $P_2$ stejný řetězec přepíše na $SS$.

$$AA \to SA \quad [A \to S]$$
$$SA \to SS \quad [A \to S]$$

$$SS \to AAS \quad [S \to AA]$$
$$AAS \to AAAA \quad [S \to AA]$$

Ze startovacího symbolu dostaneme buď terminál $a$ nebo dva startovací symboly. To znamená, že pokaždé, kdy se $AA$ přepíše na $SS$, počet terminálů $a$ se zdvojnásobí. Jazyk generovaný systémem $\Gamma$ za derivačního režimu $t$, $L(\Gamma)_t = \{a^{2^n},\ n \geq 1\}$.

### Klasifikace skupin CD jazyků

CD jazyky jsou jazyky, které jsou generovány CD gramatickými systémy. Tyto jazykové rodiny se rozdělují podle různých typů použitých CDGS. Jejich označení je

$$CD_x^y(f),$$

kde:

- $y$ určuje použití $\varepsilon$-pravidel:
  - $y = \varepsilon$ – $\varepsilon$-pravidla v komponentách jsou povolena,

- $y$ chybí – $\varepsilon$-pravidla se v komponentách nemohou vyskytovat,

- $x$ je stupeň gramatického systému:
  - $n, n \geq 1$ – gramatický systém může obsahovat maximálně $n$ komponent,
  - $\infty$ – gramatický systém nemá omezen počet komponent,

- $f$ je derivační režim, $f \in D$.

**Příklad 3.1.2.** $CD_6^{\varepsilon}(t)$ je rodina jazyků generována CD gramatickými systémy s maximálně šesti komponentami, povolenými $\varepsilon$-pravidly a pracujícími nad ukončovacím režimem.

### Hybridní CD gramatické systémy

Hybridní CD gramatické systémy nabízí možnost definovat derivační režim samostatně pro jednotlivé komponenty systému.

**Definice 3.1.5.** *Hybridní CDGS* je n-tice $\Gamma = (N, T, S, (P_1, f_1), \ldots, (P_n, f_n))$, kde:

- $N, T, P_i, S$ jsou definovány stejně jako v klasických CDGS (3.1.1),

- $f_i$ je derivační režim $i$-té komponenty, $f_i \in D$ pro všechna $i \in \{1, \ldots, n\}$.

Jazyk generovaný těmito gramatickými systémy je velmi podobný jazykům generovaným klasickými CDGS. Jediný rozdíl je v použití derivačního režimu příslušné komponenty, který je v definici označen jako $f_{j_k}$, místo společného derivačního režimu pro celý gramatický systém.

**Definice 3.1.6.** Nechť $\Gamma = (N, T, S, (P_1, f_1), \ldots, (P_n, f_n))$.
Jazyk generovaný systémem $\Gamma$,

$$L(\Gamma) = \{w \in T^* : \text{existují } v_0, v_1, \ldots, v_n \text{ takové, že } v_k \in F(G_{j_k}, v_{k-1}, f_{j_k}),$$
$$k \in \{1, \ldots, m\}, j_k \in \{1, \ldots, n\}, v_0 = S, v_m = w \text{ pro } m \geq 1\}.$$

Zápis jazykových skupin generovaných hybridními CDGS je

$$X\, CD_{x,v}^{y}(f),$$

kde:

- $x, y, f$ jsou definovány stejně jako u nehybridních CDGS,

- $v$ je nepovinné omezení počtu pravidel v komponentách:
  - $m$ – každá komponenta $P_i, i \in \{1, \ldots, n\}$ obsahuje nejvíce $m$ pravidel, $m \geq 1$,
  - $\infty$, chybí – maximální počet pravidel pro komponenty není omezen,

- $X$ určuje determinismus gramatického systému:
  - $D$ – gramatický systém je deterministický, tedy pro každé $A \to u, A \to w \in P_i$, $i \in \{1, \ldots, n\}$ platí, že $u = w$. Jinými slovy, pro všechny neterminály platí, že pro jejich přepis neexistují pravidla (ve *všech* komponentách), která by měla různé pravé strany.
  - *nic* – gramatický systém je nedeterministický. To znamená, že v *každé* komponentě existuje pravidlo pro nějaký neterminál, které má různé pravé strany.
  - $H$ – gramatický systém je hybridní a obsahuje alespoň jednu nedeterministickou komponentu a jednu deterministickou komponentu. Nezapisuje se derivační režim, který je určen samostatně pro každou komponentu.

## 3.2 Paralelní komunikující gramatické systémy

Paralelní komunikující (*parallel communicating*) – PC gramatický systém stupně $n$ je systém gramatik, v němž každá začíná vlastním startovacím symbolem (*axiomem*), které sdílí množinu neterminálů i terminálů a nově množinu komunikačních symbolů. Komunikační symboly slouží ke komunikaci na vyžádání. Kdykoliv se vyskytnou ve větné formě libovolné komponenty, proběhne *komunikační krok* (také nazýván *c-derivační krok*).

Komunikačních symbolů je stejné množství jako komponent v PC gramatickém systému, $\{Q_1, \ldots, Q_n\}$, kde index symbolu $Q_i$ odkazuje na komponentu $P_i$ [4].

**Definice 3.2.1.** Paralelní komunikující gramatický systém stupně $n, n \geq 1$ je $(n+3)$-tice

$$\Gamma = (N, K, T, (S_1, P_1), \ldots, (S_n, P_n)),$$

kde:

- $N, T$ jsou definovány stejně jako u bezkontextových gramatik (2.3.1),
- $K = \{Q_1, \ldots, Q_n\}$ je množina komunikačních symbolů ($N, K, T$ jsou navzájem disjunktní); index $i$ komunikačního symbolu $Q_i$ koresponduje s indexem $i$-té komponenty,
- $P_i, i \in \{1, \ldots, n\}$ je množina pravidel nazývané komponenty, stejně jako u CD gramatických systémů (3.1.1),
- $i$-tá gramatika systému je konstrukt $G_i = (N \cup K, T, S_i, P_i), i \in \{1, \ldots, n\}$.

**Konvence 3.2.1.** Pro označení paralelních komunikujících gramatických systémů bude dále využito zkrácené *PCGS* nebo *PC gramatické systémy.*

### Derivační kroky v PCGS

V PC gramatických systémech existují dva druhy derivačního kroku, a to $g$-derivační krok a $c$-derivační krok. První zmíněný slouží k přímé derivaci řetězce v rámci jedné komponenty bez zásahu komponent jiných a druhý slouží pro vzájemnou pomoc při derivaci řetězce. PC gramatický systém *vždy* preferuje c-derivační krok nad g-derivačním krokem. Ten se provede pokaždé, obsahuje-li alespoň jeden z řetězců $x_i, i \in \{1, \ldots, n\}$ alespoň jeden komunikační symbol.

Derivační kroky pracují nad *konfigurací* PCGS. Následující definice převzata z [18].

**Definice 3.2.2.** Nechť $\Gamma = (N, K, T, (S_1, P_1), \ldots, (S_n, P_n))$ je PC gramatický systém. Potom n-tice

$$(x_1, \ldots, x_n),\ x_i \in (N \cup K \cup T)^*,\ i \in \{1 \ldots, n\}$$

se nazývá *konfigurací* $\Gamma$. $(S_1, \ldots, S_n)$ je *počáteční konfigurací* $\Gamma$.

Nutná podmínka k proběhnutí libovolného derivačního kroku je $x_1 \notin T^*$. Pokud tato situace nastane, už je vygenerována věta jazyka definovaného PCGS a dále se s kroky nepokračuje. Více v definici 3.2.

### g-derivační krok

Nechť $\Gamma = (N, K, T, (P_1, S_1), \ldots, (P_n, S_n))$ je PCGS a $(x_1, \ldots x_n)$, $(y_1, \ldots, y_n)$ jsou konfigurace $\Gamma$. $\Gamma$ udělá g-derivační krok, formálně zapsáno

$$(x_1, \ldots, x_n) \,{}_g\Rightarrow (y_1, \ldots, y_n)$$

pokud $alph(x_i) \cap K = \emptyset$ a zároveň:

- $x_i \Rightarrow y_i$ v $G_i = (N, K, T, (S_i, P_i))$ – $y_i$ je přímo derivován z $x_i$ v gramatice $G_i$ (komponentě $P_i$), nebo
- $x_i = y_i \in T^*$ – $x_i$ již je řetězcem terminálů

pro všechna $i \in \{1, \ldots, n\}$.

### c-derivační krok

Někdy taky nazýván *komunikační krok*. Tento koncept umožňuje gramatikám spolupracovat a vzájemně si mezi sebou měnit vygenerované řetězce pomocí komunikačních symbolů. Algoritmus ukazující princip komunikačního kroku je následující:

**Algoritmus 3.2.1** c-derivační krok v PCGS

**[[pouzit Input Output a NewLine]]**

**Vstup:** konfigurace $(x_1, \ldots, x_n)$
**Výstup:** konfigurace $(y_1, \ldots, y_n)$

**for all** $i \in \{1, \ldots, n\}$ **do**
    $z_i \leftarrow x_i$
**end for**
**for all** $i \in \{1, \ldots, n\}$ **do**
    **if** $alph(x_i) \cap K \neq \emptyset$ **and foreach** $Q_j$ **in** $x_i$: $alph(x_j) \cap K = \emptyset$ **then**
        **for all** $Q_j$ **in** $x_i$ **do**
            $z_j \leftarrow S_j$ ▷ vynecháno, pokud PCGS pracuje na nevracejícím se režimu
            zaměň $Q_j$ za $x_j$ v $x_i$
            $z_i \leftarrow x_i$ ▷ $x_i$ = řetězec, který vznikl o krok zpět
        **end for**
    **end if**
**end for**
proveď $(x_1, \ldots, x_n) \,{}_c\Rightarrow (y_1, \ldots, y_n)$ s $y_i = z_i,\ i \in \{1, \ldots, n\}$

## Přímá derivace v PCGS

**Definice 3.2.3.** Konfigurace $(x_1, \ldots, x_n)$ přímo derivuje konfiguraci $(y_1, \ldots, y_n)$, zapsáno

$$(x_1, \ldots, x_n) \Rightarrow (y_1, \ldots, y_n)$$

právě tehdy, když

$$(x_1, \ldots, x_n) \;{}_g\Rightarrow (y_1, \ldots, y_n)$$

nebo

$$(x_1, \ldots, x_n) \;{}_c\Rightarrow (y_1, \ldots, y_n).$$

### Jazyk generovaný PCGS

Generování větné formy končí v momentě, kdy první komponenta dosáhne řetězce terminálů a na řetězcích ostatních komponent nezáleží.

**Definice 3.2.4.** Nechť $\Gamma = (N, K, T, (P_1, S_1), \ldots, (P_n, S_n))$ je PCGS. Jazyk generovaný $\Gamma$ je stejný jako jazyk generovaný jeho první komponentou:

$$L_f(\Gamma) = \{w \in T^* : (S_1, \ldots, S_n) \Rightarrow_f^* (w, \alpha_2, \ldots, \alpha_n), \\ for\ \alpha_i \in \{N \cup T \cup K\},\ i \in \{2, \ldots, n\},\ f \in \{r, nr\}\},$$

kde $r$ a $nr$ specifikuje *vracející* nebo *nevracející* PCGS.

### Vracející a nevracející se režim

Pokud PC gramatický systém pracuje na vracejícím se režimu, potom komponenty, které v rámci komunikačního kroku poslaly svůj řetězec jiným komponentám, generují řetězec od svého axiomu. Při nevracejícím se režimu komponenty pokračují ve zpracovávání aktuálního řetězce.

Tato skutečnost se projeví na samotném komunikačním kroku, u kterého se vynechá přiřazení axiomu do řetězce $z_j$ – neprovede se krok na řádku 10 algoritmu 3.2.1.

### Centralizované PCGS

Centralizované PC gramatické systémy mají tu vlastnost, že pouze první komponenta (nazývaná *master*) systému může generovat komunikační symboly a tím žádat ostatní komponenty o řetězce. Řeší jeden z možných případů uváznutí, kdy komponenty v cyklu zavádějí komunikační symboly a donekonečna se provádí stejná sekvence komunikačních kroků.

**Definice 3.2.5.** Nechť $\Gamma = (N, K, T, (S_1, P_1), \ldots, (S_n, P_n))$ je PCGS. Pokud pouze $P_1$ může uvést komunikační symboly, formálně

$$P_i \subseteq (N \cup T)^* \times (N \cup T)^* \text{ pro } i \in \{2, \ldots, n\},$$

potom $\Gamma$ je centralizovaný PC gramatický systém. Jinak je necentralizovaný.

### Příklady

V prvním příkladu bude pouze demonstrován princip obou derivačních kroků na systému, který generuje jen velmi omezený jazyk.

**Příklad 3.2.1.** Nechť $\Gamma = (\{S_1, S_2, S_3\}, \{Q_3\}, \{a, b\}, (S_1, \{S_1 \to Q_3\}), (S_2, \{S_2 \to a\}), (S_3, \{S_3 \to b\}))$ je PCGS. Počáteční konfigurace $\Gamma$ je zřejmě $(S_1, S_2, S_3)$. Víme, že PC gramatické systémy *vždy* preferují c-krok nad g-krokem, je-li to možné. Nutná podmínka je, aby alespoň jeden z řetězců v aktuální konfiguraci obsahoval alespoň jeden komunikační symbol, což aktuálně není splněno. $\Gamma$ tedy udělá g-krok.

$$(S_1, S_2, S_3) \ {}_g\Rightarrow\ (Q_3, a, b)$$

Nyní je v konfiguraci komunikační symbol, což indikuje, že bude následovat komunikační krok. Při postupu podle algoritmu 3.2.1 je postup následující:

1. Zavedení pomocných řetězců $z_1 = Q_3$, $z_2 = a$, $z_3 = b$ podle řádků 4 – 6.

2. Kontrola, zda řetězec $x_i$, $i \in \{1, \ldots, n\}$ z původní konfigurace obsahuje komunikační symboly (podmínka $alph(x_i) \cap K \neq \emptyset$).

   - V tomto příkladu splňuje podmínku pouze řetězec $x_1$, který obsahuje $Q_3$.

3. Pokud $x_i$ obsahuje komunikační symboly $Q_j$, z každého se přečte index $j$ a proběhne kontrola, zda všechny $j$-té řetězce konfigurace ($x_j$) *neobsahují* komunikační symboly. Tato část koresponduje s podmínkou **foreach** $Q_j$ **in** $x_i : alph(x_i) \cap K = \emptyset$ na řádku 8.

   - Řetězec $x_1$ obsahuje jeden komunikační symbol $Q_3$, jehož index odkazuje na řetězec $x_3$ z aktuální konfigurace. Hodnota řetězce $x_3$ je $b$, ten žádné další komunikační symboly neobsahuje.

4. Každý $Q_j$ v $x_i$ se nahradí za $x_j$, pokud prošel podmínkou v předchozím kroku. Zároveň se $z_j$ nastaví na počáteční symbol $j$-té komponenty. Tyto kroky jsou v algoritmu na řádcích 9 – 12.

   - Komunikační symbol $Q_3$ bude v $x_1$ nahrazen řetězcem $b$ z $x_3$. Dále $x_3 \leftarrow S_3$ a $z_1 \leftarrow x_1$. Aktuálně $z_1 = b$, $z_2 = a$, $z_3 = S_3$.

5. Proveď $(x_1, \ldots, x_n) \ {}_c\Rightarrow (y_1, \ldots, y_n)$, kde $y_i = z_i$ pro $i \in \{1, \ldots, n\}$.

   - Nová konfigurace $(y_1, y_2, y_3)$ bude stejná, jako pomocná konfigurace $(z_1, z_2, z_3)$, a to $(b, a, S_3)$.

$\Gamma$ provede komunikační krok z $(Q_3, a, b)$ do $(b, a, S_3)$.

$$(Q_3, a, b) \ {}_c\Rightarrow (b, a, S_3)$$

Další kroky už $\Gamma$ provádět nebude, protože řetězec generovaný první komponentou je již řetězec terminálních symbolů. V tomto příkladu byla použita sémantika vracejícího se PCGS, nicméně při použití nevracejícího by jazyk vypadal stejně, jen výsledná konfigurace by se lišila v řetězci $x_3$.

$$L(\Gamma)_r = L(\Gamma)_{nr} = \{b\}$$

Ve druhém příkladu bude demonstrováno několik derivačních kroků a bude se zkoumat výsledný generovaný jazyk.

**Příklad 3.2.2.** Nechť $\Gamma = (\{S_1, S_1', S_2, S_3\}, K, a, b, c, (S_1, P_1), (S_2, P_2), (S_3, P_3))$, kde:

$$\begin{aligned} P_1 &= \{S_1 \to abc, S_1 \to a^2b^2c^2, S_1 \to aS_1', S_1 \to a^3Q_2, \\ &\qquad S_1' \to aS_1', S_1' \to a^3Q_2, S_2 \to b^2Q_3, S_3 \to c\}, \\ P_2 &= \{S_2 \to bS_2\}, \\ P_3 &= \{S_3 \to cS_3\}. \end{aligned}$$

Komunikační symboly může uvést pouze komponenta $P_1$, a proto se jedná o *centralizovaný* PC gramatický systém.

Počáteční konfigurace je $(S_1, S_2, S_3)$. $\Gamma$ může generovat konfiguraci $(aS_1', bS_2, cS_3)$ za použití pravidel $(S_1 \to aS_1', S_2 \to bS_2, S_3 \to cS_3)$.

$$(S_1, S_2, S_3) \Rightarrow (aS_1', bS_2, cS_3)$$

Zřejmě při použití pravidel $(S'_1 \to aS'_1, S_2 \to bS_2, S_3 \to cS_3)$ $n$-krát po sobě je možné generovat konfigurace $(a^nS'_1, b^nS_2, c^nS_3),\ n \geq 1$.

$$(aS'_1, bS_2, cS_3) \Rightarrow^* (a^nS'_1, b^nS_2, c^nS_3)$$

Je vidět, že komponenty $P_2$ a $P_3$ neustále používají ta samá pravidla, což je logické vzhledem k jejich kardinalitě. Pro zjednodušení zápisu, při každém g-derivačním kroku bude zmíněno pouze pravidlo komponenty $P_1$, pravidla komponent $P_2$ a $P_3$ budou vždy stejná.

Z konfigurace $(a^nS'_1, b^nS_2, c^nS_3)$ se na rozdílný výsledek dá dostat pouze za použití pravidla $S'_1 \to a^3Q_2$.

$$(a^nS'_1, b^nS_2, c^nS_3) \Rightarrow (a^{n+3}Q_2, b^{n+1}S_2, c^{n+1}S_3)$$

Provedeme c-derivační krok a zaměníme $Q_2$ za $x_2$, nový $x_2$ bude $S_2$, který je axiomem $G_2$.

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

Jediné pravidlo pro symbol $S_2$ v komponentě $P_1$ je $S_2 \to b^2Q_3$.

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

Zaměníme $Q_3$ za $x_3$ nový $x_3$ bude $S_3$, který je axiomem $G_3$.

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

Pro $S_3$ je v $P_1$ také pouze jediné pravidlo, které se může aplikovat, a to $S_3 \to c$.

$$(a^{n+3}b^{n+3}c^{n+2}S_3, bS_2, S_3) \Rightarrow (a^{n+3}b^{n+3}c^{n+3}, bbS_2, cS_3)$$

$x_1$ je aktuálně řetězec terminálů, tudíž na něj nelze aplikovat žádné další pravidlo a neobsahuje komunikační symboly. Je tedy generovaným jazykem pro $\Gamma$ ve vracejícím se režimu. Takto by vypadaly kroky pro nevracející se režim (konfigurace se začnou lišit od výskytu komunikačních symbolů):

$$\begin{aligned}(a^{n+3}Q_2, b^{n+1}S_2, c^{n+1}S_3) &\Rightarrow (a^{n+3}b^{n+1}S_2, b^{n+1}S_2, c^{n+1}S_3)\\ &\Rightarrow (a^{n+3}b^{n+3}Q_3, b^{n+2}S_2, c^{n+2}S_3)\\ &\Rightarrow (a^{n+3}b^{n+3}c^{n+2}S_3, b^{n+2}S_2, c^{n+2}S_3)\\ &\Rightarrow (a^{n+3}b^{n+3}c^{n+3}, b^{n+3}S_2, c^{n+3}S_3)\end{aligned}$$

Je tedy zřejmé, že:

$$L(\Gamma)_r = L(\Gamma)_{nr} = \{a^nb^nc^n,\ n \geq 1\}.$$

# Kapitola 4

# Syntaktická analýza

Tato kapitola se zabývá syntaktickou analýzou, která provádí kontrolu, zda je vstupní řetězec v jazyce, který je formálně definován. Ostatní části překladačů v této práci nebudou rozebírány, nebude-li to vyžadovat kontext. Celý proces překladu programů je podrobně popsán v [9], odkud bude do této kapitoly převzato největší množství informací.

Syntax jazyka, ve kterém je napsán zdrojový kód, je nejčastěji popsána gramatikou (v praxi nejčastěji bezkontextovou) jazyka s konečnou množinou pravidel. Pomocí těchto pravidel analyzátor zkontroluje, že posloupnost *tokenů*, které jsou na vstupu od lexikálního analyzátoru, je korektní, podle použitých pravidel v gramatice. Při tomto procesu syntaktický analyzátor generuje *derivační strom*, kde každý uzel a jeho potomci reprezentují použité pravidlo. Tento proces může probíhat *shora dolů*, čímž se zabývá kapitola 4.1, případně *zdola nahoru*, čímž se zabývá kapitola 4.2. Tyto názvy jsou odvozeny od směru tvorby derivačního stromu, kdy analýza shora dolů postupuje od kořene k listům, analýza zdola nahoru naopak.

Než se začneme zabývat těmito dvěma druhy syntaktické analýzy, pojďme si definovat pojmy *nejlevější a nejpravější derivace* a jejich sekvence.

**Definice 4.0.1.** Nechť $G = (N, T, P, S)$ je BKG, $r : A \to \alpha \in R$, $u \in T^*$, $\gamma \in (N \cup T)^*$.
$G$ provede nejlevější derivační krok (*leftmost derivation step*) z $uA\gamma$ do $u\alpha\gamma$, podle pravidla $r$, symbolicky zapsáno

$$uA\gamma \ {}_{lm}\Rightarrow u\alpha\gamma \ [r].$$

$G$ provede nejpravější derivační krok (*rightmost derivation step*) z $\gamma Au$ do $\gamma\alpha u$, podle pravidla $r$, symbolicky zapsáno

$$\gamma Au \ {}_{rm}\Rightarrow \gamma\alpha u \ [r].$$

**[[ciste typograficky, vadi zalomene rovnice?]]**

**Definice 4.0.2.** Nechť $G = (N, T, P, S)$ je BKG a $\alpha_0, \alpha_1, \ldots, \alpha_n$ je sekvence, kde $\alpha_i \in (N \cup T)^*$, $i \in \{1, \ldots, n\}$.
Pokud $\alpha_{j-1} \ {}_{lm}\Rightarrow \alpha_j \ [r_j]$ v $G$, kde $r_j \in R$, $j \in \{1, \ldots, n\}$, pak $G$ provede sekvenci nejlevějších derivačních kroků (*nejlevější derivaci*) z $\alpha_0$ do $\alpha_n$ podle $r_1, r_2, \ldots, r_n$, symbolicky zapsáno jako

$$\alpha_0 \ {}_{lm}\Rightarrow \alpha_n \ [r_1 r_2 \ldots r_n].$$

Pokud $\alpha_{j-1} \ {}_{rm}\Rightarrow \alpha_j \ [r_j]$ v $G$, kde $r_j \in R$, $j \in \{1, \ldots, n\}$, pak $G$ provede sekvenci nejpravějších derivačních kroků (*nejpravější derivaci*) z $\alpha_0$ do $\alpha_n$ podle $r_1, r_2, \ldots, r_n$, symbolicky zapsáno jako

$$\alpha_0 \ {}_{rm}\Rightarrow \alpha_n \ [r_1 r_2 \ldots r_n].$$

[[jestli zbude cas, napsat vlastnimi slovy, inspirace v [9], strana 83]]

## 4.1 Syntaktická analýza shora dolů

Uvažujme BKG $G$. Syntaktický analyzátor pracující shora dolů pro $G$ je reprezentován zásobníkovým automatem $M$, který je ekvivalentní k $G$. $M$ na svém zásobníku simuluje nejlevější derivace a tvorbu derivačního stromu pro řetězce generované gramatikou $G$. Aby tento model fungoval, musí $M$ obsahovat korespondující pravidla pro pravidla $G$ a navíc mít pomocné pravidlo pro mazání terminálních symbolů z vrchou zásobníku, pokud přečte stejný symbol ze vstupní pásky.

Pokud na vrcholu zásobníku je neterminál $A$, pak $M$ simuluje nejlevější derivační krok, který by udělala gramatika $G$ pomocí pravidla $r : A \to \alpha \in R$. Provede expanzi pravidla nahrazením neterminálu $A$ z vrcholu zásobníku za $reversal(\alpha)$.

**Algoritmus 4.1.1** Převod BKG na ZA

[[pouzit Input Output a NewLine]]

1: **Vstup:** BKG $G = (N, T, P, S)$
2: **Výstup:** ZA $M = (Q, \Sigma, \Gamma, R, s, \#, F)$
3:
4: $Q \leftarrow \{s, f\}$
5: $\Sigma \leftarrow T$
6: $\Gamma \leftarrow N \cup T \cup \{\#\}$
7: **for all** $a \in \Sigma$ **do**
8: přidej $sa \to as$ do $R$
9: **end for**
10: **for all** $A \to \alpha \in R$ **do**
11: přidej $\alpha s \to As$ do $R$
12: **end for**
13: přidej $\#Ss \to f$ do $R$
14: $F \leftarrow \{f\}$

Algoritmus 4.1.1 uvádí konstrukci zásobníkového automatu. Implementace tohoto systému vyžaduje implementaci takového automatu pomocí nějaké variace zásobníku. Existují dvě metody, a to *prediktivní syntaktická analýza* a *rekurzívní sestup*. Více k těmto metodám je psáno na konci této kapitoly.

### Prediktivní množiny

Předpokládejme BKG $G$ a příslušný ZA $M$ zkonstruovaný podle algoritmu 4.1.1. Dále předpokládejme, že existuje neterminál $A \in N$ takový, že pro něj existují dvě pravidla ve tvaru $A \to X\alpha$ a $A \to Y\beta$. $A$ musí být v dalším kroku přepsán za nový řetězec a zároveň musí být deterministicky zvolené pravidlo, které se použije.

K deterministickému výběru slouží množina $Predict(A \to \gamma)$. Pokud pravidla, která na levé straně mají neterminál $A$, mají množinu $Predict(A \to \gamma)$ navzájem disjunktní, pravidlo je zvoleno deterministicky. Pro konstrukci množin $predict(A \to \gamma)$ je zapotřebí množin $Empty(\gamma)$, $First(\gamma)$ a $Follow(A)$.

Definice a algoritmy v této kapitole převzaty z [11, 6], kde jsou popsány velmi intuitivně, nicméně ne tak podrobně, jako například v [9]. Inspirace pro strukturu této kapitoly a slovní popisy algoritmů čerpána z [6].

Množina $Empty(\alpha)$ je množina, která obsahuje jediný prvek $\varepsilon$, pokud $\alpha$ derivuje $\varepsilon$, jinak je prázdná. Jinými slovy, pokud aktuální řetězec je možné vymazat pomocí $\varepsilon$-pravidel, pak bude množina pro tento řetězec obsahovat jediný prvek $\varepsilon$, jinak je prázdná.

**Definice 4.1.1.** Nechť $G = (N, T, P, S)$ je BKG.
Množina $Empty(\alpha)$, pro každé $\alpha \in N \cup T$, je definována jako

$$\begin{aligned} &1) \quad Empty(\alpha) = \{\varepsilon\} \text{ pokud } \alpha \Rightarrow^* \varepsilon, \\ &2) \quad Empty(\alpha) = \emptyset \text{ jinak.} \end{aligned}$$

**[[nejaky obrazecek by to chtelo]]**

**Algoritmus 4.1.2** Množina $Empty(\alpha)$

**[[pouzit Input Output a NewLine]]**

1: **Vstup:** BKG $G = (N, T, P, S)$
2: **Výstup:** $Empty(\alpha)$ pro každý symbol $\alpha \in N \cup T$
3:
4: **for all** $a \in T$ **do**
5: $\quad Empty(a) \leftarrow \emptyset$
6: **end for**
7: **for all** $A \in N$ **do**
8: $\quad$ **if** $A \to \varepsilon \in P$ **then**
9: $\quad\quad Empty(A) \leftarrow \{\varepsilon\}$
10: $\quad$ **else**
11: $\quad\quad Empty(A) \leftarrow \emptyset$
12: $\quad$ **end if**
13: **end for**
14: **while** je možné měnit nějakou množinu $Empty(A)$ **do**
15: $\quad$ **if** $A \to X_1X_2 \dots X_n \in P$ **and** $Empty(X_i) = \{\varepsilon\}$ **foreach** $i \in \{1, \dots, n\}$ **then**
16: $\quad\quad Empty(A) \leftarrow \{\varepsilon\}$
17: $\quad$ **end if**
18: **end while**

Na začátku výpočtu množin $Empty(\alpha)$ je pro každý terminál nastavena tato množina na prázdnou, protože z terminálů nikdy prázdný řetězec nedostaneme. Dále, pokud je možné přímo derivovat prázdný řetězec z aktuálního neterminálu $A$, pak je zřejmé, že neterminál $A$ bude vymazán, pokud pravidlo pro prázdný řetězec bude použito. Nakonec se algoritmus znovu dívá na všechny řetězce $\alpha$ a jejich množiny $Empty(\alpha)$, které dále mohou být měněny. Jsou to takové neterminály $A$, které mají množinu $Empty(A)$ neprázdnou, v druhém případě se množina $Empty(A)$ již nezmění.

V následujícím algoritmu si představíme množinu $Empty(X_1X_2 \dots X_n)$. Z výše psaného textu vyplývá, je třeba tuto možnost řešit výhradně pro neterminály, proto budeme v algoritmu označovat symboly, pro které se konstruuje množina $Empty(X)$ jako neterminály podle konvence 2.2.2. Tento algoritmus pomůže při definici dalších množin, nicméně není k tomu nutný.

**Algoritmus 4.1.3** Množina $Empty(X_1X_2\dots X_n)$

**[[pouzit Input Output a NewLine]]**

**Vstup:** BKG $G = (N, T, P, S)$ a $Empty(X)$ pro všechny symboly $X \in N \cup T$
**Výstup:** $Empty(X_1X_2\dots X_n)$, kde $(X_1X_2\dots X_n) \in (N \cup T)^+$

**if** $Empty(X_i) = \varepsilon$ **foreach** $i \in \{1, \dots, n\}$ **then**
    $Empty(X_1X_2\dots X_n) \leftarrow \{\varepsilon\}$
**else**
    $Empty(X_1X_2\dots X_n) \leftarrow \emptyset$
**end if**

Množina $First(\alpha)$ je výčet všech terminálů, kterými může začínat řetězec derivovatelný z $\alpha$.

**Definice 4.1.2.** Nechť $G = (N, T, P, S)$ je BKG.
Pro každé $\alpha \in (N \cup T)^*$ je definováno $First(\alpha)$ jako

$$First(\alpha) = \{a : a \in T,\ \alpha \Rightarrow^* a\beta,\ \beta \in (N \cup T)^*\}.$$

**[[ilustrujici obrazek (podobny jako v ifj prezentaci 7)]]**

**Algoritmus 4.1.4** Množina $First(\alpha)$

**[[pouzit Input Output a NewLine]]**

**Vstup:** $G = (N, T, P, S)$
**Výstup:** $First(\alpha)$ pro každé $\alpha \in (N \cup T)^*$

**for all** $a \in T$ **do**
    $First(a) \leftarrow \{a\}$
**end for**
**for all** $A \in N$ **do**
    $First(A) \leftarrow \emptyset$
**end for**
**while** je možné měnit nějakou množinu $First(A)$ **do**
    **if** $A \rightarrow X_1X_2\dots X_{k-1}X_k\dots X_n \in P$ **then**
        $First(A) \leftarrow First(A) \cup First(X_1)$
        **if** $Empty(X_1X_2\dots X_{k-1}) = \{\varepsilon\}$ **then**
            $First(A) \leftarrow First(A) \cup First(X_k)$
        **end if**
    **end if**
**end while**

Je zřejmé, že řetězec derivovatelný z terminálů $a$ musí určitě začínat terminálem $a$. Ve druhém kroku algoritmu nastavíme $First(A)$ na prázdnou množinu, bude se měnit až v dalších krocích. Řetězec derivovatelný z neterminálu $A$ pak může začínat:

1. terminály z množiny $First(\alpha_1)$, ať už se jedná o terminál či neterminál,
2. terminály z množiny $First(\alpha_k)$, pokud řetězec neterminálů $X_1X_2\dots X_{k-1}$ může být vymazán, respektive $Empty(X_1X_2\dots X_{k-1}) = \varepsilon$.

Pro snadnější definici množiny $Follow(A)$ si uveďme algoritmus množiny $First(\alpha)$ pro neprázdné řetězce.

**Algoritmus 4.1.5** Množina $First(\alpha_1\alpha_2\dots\alpha_n)$

**Vstup:** BKG $G = (N, T, P, S)$ a $Empty(\alpha), First(\alpha)$ pro všechny symboly $\alpha \in N \cup T$
**Výstup:** množiny $First(\alpha_1\alpha_2\dots\alpha_n)$, kde $(\alpha_1\alpha_2\dots\alpha_n) \in (N \cup T)^+$

1: $First(\alpha_1\alpha_2\dots\alpha_n) \leftarrow First(\alpha_1)$
2: **while** je možné měnit nějakou množinu $First(\alpha_1\alpha_2\dots\alpha_{k-1}\alpha_k\dots\alpha_n)$ **do**
3: **if** $Empty(\alpha_1\alpha_2\dots\alpha_{k-1}) = \varepsilon$ **then**
4: $First(\alpha_1\alpha_2\dots\alpha_n) \leftarrow First(\alpha_1\alpha_2\dots\alpha_n) \cup First(\alpha_k)$
5: **end if**
6: **end while**

Další velmi podstatnou množinou, kterou syntaktický analyzátor potřebuje, je množina $Follow(A)$. Ta určuje, jaké symboly se mohou vyskytovat za neterminálem $A$. Nutnost této znalosti vyplývá z možnosti přepsat neterminály, pro které platí $Empty(A) = \{\varepsilon\}$, na prázdný řetězec. Kdyby analyzátor tuto množinu neměl k dispozici, neměl by v těchto případech jak zjistit, zda je aktuální symbol na vstupu korektní nebo ne. K definici množin $Follow(A)$ se používá pomocný symbol \$, který reprezentuje konec vstupního řetězce.

**Definice 4.1.3.** Nechť $G = (N, T, P, S)$ je BKG.
Pro všechny neterminály $A \in N$ definujeme množinu $Follow(A)$ jako

$$Follow(A) = \{a : a \in T,\ S \Rightarrow^* \alpha A a \beta,\ \alpha, \beta \in (N \cup T)^*\} \cup \{\$ : S \Rightarrow^* \alpha A,\ \alpha \in (N \cup T)^*\}.$$

**Algoritmus 4.1.6** Množina $Follow(A)$

**Vstup:** BKG $G = (N, T, P, S)$
**Výstup:** $Follow(A)$ pro každé $A \in N$

1: $Follow(S) \leftarrow \$$
2: **while** je možné měnit nějakou množinu $Follow(A)$ **do**
3: **if** $A \to \alpha B \beta \in R$ **then**
4: **if** $\beta \neq \varepsilon$ **then**
5: $Follow(B) \leftarrow Follow(B) \cup First(\beta)$
6: **end if**
7: **if** $Empty(\beta) = \{\varepsilon\}$ **then**
8: $Follow(B) \leftarrow Follow(B) \cup Follow(A)$
9: **end if**
10: **end if**
11: **end while**

Z uvedené definice 4.1.3 a algoritmu 4.1.6 vyplývá, že v množině $Follow(A)$ se mohou vyskytovat:

1. terminální symboly vyskytující se za neterminálem $A$ v libovolném zderivovaném řetězci,

2. symbol ukončující vstupní řetězec, pokud za neterminálem $A$ již nebudou žádné další neterminály.

Množina $Follow(A)$ byla poslední množina, která byla třeba definovat, abychom mohli definovat množinu $Predict(A \to \alpha)$, ze které se poté může zkonstruovat *LL tabulka*. Množina $Predict(A \to \alpha)$ už je prakticky dobře použitelná pro syntaktickou analýzu – je to množina všech terminálů, které mohou být aktuálně nejlevěji vygenerovány, pokud pro libovolnou větnou formu použijeme pravidlo $A \to \alpha$.

**Definice 4.1.4.** Nechť $G = (N, T, P, S)$ je BKG.
Pro každé pravidlo $A \to \alpha \in P$ definujeme množinu $Predict(A \to \alpha)$ jako

$$Predict(A \to \alpha) = First(\alpha) \cup Follow(A)$$

pokud $Empty(\alpha) = \{\varepsilon\}$,

$$Predict(A \to \alpha) = First(\alpha)$$

pokud $Empty(\alpha) = \emptyset$.

Prakticky díky této množině je syntaktický analyzátor schopen deterministicky vybrat pravidlo, které se aplikuje. Mějme dvě pravidla, $p$ a $q$, mezi kterými se analyzátor rozhoduje a nechť na vstupu je terminál $a$. Pokud $a \in Predict(p)$, pak se vybere pravidlo $p$, pokud $a \in Predict(q)$, pak se vybere pravidlo $q$. Jak už bylo zmiňováno na začátku této kapitoly, všechny množiny $Predict(p)$ musí být navzájem disjunktní, jinak by výběr byl opět nedeterministický. Tento koncept je aplikován v *LL(1) gramatikách*.

### LL gramatiky

Výše uvedené množiny pracují s gramatikami, které se nazývají *LL(1) gramatiky*. Obecně gramatiky mohou být *LL(k)*. To jsou gramatiky, u kterých stačí $k$ vstupních tokenů k tomu, aby jejich analyzátor mohl deterministicky vybrat pravidlo pro aktuální neterminál. Obecně platí, že LL(1) gramatiky jsou slabší (generují podmnožinu jazyků) než BKG [9]. LL(1) gramatika je definována následovně:

**Definice 4.1.5.** Nechť $G = (N, T, P, S)$ je BKG.
$G$ je *LL(1) gramatikou*, pokud pro libovolná dvě pravidla $p, q \in P$ platí

$$p \neq q \text{ a zároveň } Predict(p) \cap Predict(q) = \emptyset.$$

Jinými slovy neexistuje žádné $a \in T$ a $A \in N$ takové, že $A \to \alpha, A \to \beta \in P$ a zároveň $a \in First(\alpha)$ a $a \in First(\beta)$.

**Konvence 4.1.1.** Tato práce se zabývá především LL(1) gramatikami. Pro zjednodušení, LL(1) gramatiky budou implicitně označovány jako *LL gramatiky*. Bude-li se jednat o LL($k$), bude to explicitně zapsáno.

Některé bezkontextové gramatiky se dají převést na ekvivalentní LL gramatiky pomocí technik *faktorizace* (vytýkání) a *odstranění levé rekurze*.

Myšlenka faktorizace je vytknutí stejné sekvence terminálů, která je společným prefixem několika pravých stran pravidel a nahrazení zbývajícího řetězce za nový neterminál.

Z nového neterminálu dále vznikají nová pravidla. Například, mějme tato pravidla:

$$\begin{aligned} A &\to a\alpha_1 \\ A &\to a\alpha_2 \\ &\vdots \\ A &\to a\alpha_n \end{aligned}$$

Terminál $a$ se může vytknout a místo $\alpha_1, \alpha_2, \ldots, \alpha_n$ se vloží nový neterminál $B$, ze kterého se dále generují $\alpha_1, \alpha_2, \ldots, \alpha_n$.

$$\begin{aligned} A &\to aB \\ B &\to \alpha_1 \\ B &\to \alpha_2 \\ &\vdots \\ B &\to \alpha_n \end{aligned}$$

Levá rekurze je případ, kdy stejný neterminál, jako je na levé straně pravidla, je zároveň nejlevějším symbolem řetězce na pravé straně pravidla, například:

$$\begin{aligned} A &\to A\alpha \\ A &\to a \end{aligned}$$

Tato pravidla se dají přepsat do tvaru:

$$\begin{aligned} A &\to aB \\ B &\to \alpha B \\ B &\to \varepsilon \end{aligned}$$

**[[opet by to chtelo nejaky obrazky]]**

### LL tabulka a její konstrukce

*LL tabulka* je abstrakcí k množinám $Predict(p)$ jednotlivých pravidel LL gramatiky $G = (N, T, P, S)$. Řádky tabulky jsou indexovány neterminály z množiny $N$, sloupce potom terminály z množiny $T \cup \{\$\}$. Samotné položky tabulky jsou čísla pravidel, která byla přidělena funkcí mapující dvojici $(A, a)$, $a \in T \cup \{\$\}$, $A \in N$ na

- pravidlo, respektive symbol reprezentující konkrétní pravidlo,
- symbol vyjadřující neexistenci pravidla pro danou dvojici; v tomto případě syntaktická analýza zahlašuje chybu.

Jinak řečeno, nechť $\alpha(A, a)$ reprezentuje políčko v LL tabulce a nechť $p : A \to X_1 X_2 \ldots X_n \in P$. Pokud $a \in First(X_1)$, pak $\alpha(A, a) = p$, jinak nastává chyba. Následující příklad převzat z [9].

**Příklad 4.1.1.** Nechť $G = (N, T, P, S)$ je LL gramatika a nechť má sestavené množiny $Predict(p)$ pro každé $p \in P$.
Pravidla a jejich množiny $Predict$ nechť jsou definovány následovně:

| Pravidlo $r \in R$ | $Predict(r)$ |
|---|---|
| $E \to TA$ | $i, ($ |
| $A \to \vee\, TA$ | $\vee$ |
| $A \to \varepsilon$ | $), \$$ |
| $T \to FB$ | $i, ($ |
| $B \to \wedge\, FB$ | $\wedge$ |
| $B \to \varepsilon$ | $\vee, )$ |
| $T \to F$ | $i, ($ |
| $T \to (E)$ | $($ |
| $F \to i$ | $i$ |

Tabulka 4.1.1: Pravidla $G$ a jejich množiny $Predict$.

Nechť symbol $\times$ reprezentuje prázdné políčko LL tabulky. LL tabulka pro pravidla a jejich $Predict$ množiny z tabulky 4.1.1, zkonstruována podle výše popsané metody, vypadá takto:

| $i$ | $i$ | $\vee$ | $\wedge$ | $($ | $)$ | $\$$ |
|---|---|---|---|---|---|---|
| $E$ | 1 | $\times$ | $\times$ | 1 | $\times$ | $\times$ |
| $A$ | $\times$ | 2 | $\times$ | $\times$ | 3 | 3 |
| $T$ | 4 | $\times$ | $\times$ | 4 | $\times$ | $\times$ |
| $B$ | $\times$ | 6 | 5 | $\times$ | 6 | 6 |
| $F$ | 9 | $\times$ | $\times$ | 8 | $\times$ | $\times$ |

Tabulka 4.1.2: LL tabulka pro pravidla z tabulky 4.1.1.

### Prediktivní syntaktická analýza

*Prediktivní syntaktická analýza* a *rekurzívní sestup* jsou dvě metody, pomocí kterých lze implementovat syntaktický analyzátor. Jak je zmíněno na začátku této kapitoly, je potřeba nějakým způsobem provádět analýzu pomocí zásobníkového automatu. Při použití prediktivní analýzy je nutné tento zásobník implementovat explicitně, rekurzívní sestup si zásobník tvoří v pozadí automaticky díky volání funkcí pro každý neterminál. Součástí této práce je implementace explicitního zásobníkového automatu, proto nebude rekurzívní sestup podrobně popsán; této metodě se detailně věnuje autor v [9].

Výhodou implementace prediktivní analýzy s LL tabulkou je jediná implementace algoritmu. V případě změny pravidel v gramatice je nutné pouze změnit LL tabulku, odkud automat odebírá pravidla, která má použít. Oproti tomu v rekurzívním sestupu je nutné měnit celé funkce. Algoritmus prediktivní syntaktické analýzy je popsán v algoritmu 4.1.7.

## 4.2 Syntaktická analýza zdola nahoru

Syntaktická analýza zdola nahoru provádí konstrukci derivačního stromu zleva doprava a zespodu nahoru, tudíž jde od listových uzlů (tokenů) směrem nahoru ke kořeni (výchozí neterminál) a provádí nejpravější derivaci vstupního řetězce. Každý krok této metody je reprezentován operacemi posunu (*shift*) nebo redukce (*reduce*). Operace shift provádí vložení symbolu na zásobník a redukce simuluje nejpravější derivační krok. Během redukce ana-

**Algoritmus 4.1.7** Prediktivní syntaktická analýza založená na LL tabulce

**Vstup:** LL tabulka pro $G = (N, T, P, S)$, řetězec $x \in T^*$

**Výstup:** Levý rozbor pro $x$, pokud $x \in L(G)$, jinak chyba.

```
1:  push($) and push(S)
2:  repeat
3:      X ← stack.top()
4:      a ← aktuální token ze vstupního řetězce
5:      if X = $ then
6:          if a = $ then
7:              úspěch
8:          else
9:              chyba
10:         end if
11:     else if X ∈ T then
12:         if X = a then
13:             stack.pop(X)
14:             a ← nový token ze vstupního řetězce
15:         else
16:             chyba
17:         end if
18:     else if X ∈ N then
19:         if r : X → α ∈ LL table[X, a] then
20:             stack.pop(X)
21:             stack.top() ← reversal(α)
22:         else
23:             chyba
24:         end if
25:     end if
26: until úspěch or chyba
```

lyzátor musí na zásobníku najít pravou stranu libovolného pravidla (nazývanou *handle*), zredukovat jej a výsledný neterminál vložit zpět na zásobník.

Pro tuto práci je důležitá pouze jedna z metod analýzy zdola nahoru, a to *precedenční analýza výrazů*. Existují další metody, jako je například *LR syntaktická analýza*, ta ale není předmětem této práce, popsána bude pouze precedenční analýza. LR analýza je podrobně popsána v [9].

### Precedenční analýza výrazů

Precedenční analyzátor pracuje na základě $G = (N, T, P, S)$ pomocí *precedenční tabulky*. Precedenční tabulka je struktura se sloupci i řádky tvořenými symboly z množiny $T \cup \{\$\}$, kterými se tabulka v praxi indexuje. Každé pole precedenční tabulky má hodnotu jednoho ze symbolů množiny $\{<, >, =, \times, \checkmark\}$.

Analyzátor si na zásobník ukládá terminály i neterminály a navíc speciální znak $<$, který určuje počátek řetězce připravený na redukci. Dno zásobníku pak reprezentuje symbol \$; zásobníková abeceda toho analyzátoru je tedy $T \cup \{<, \$\}$.

Než bude ukázán algoritmus pro precedenční analýzu, je třeba zadefinovat operace *SHIFT* a *REDUCE*, se kterými analyzátor pracuje.

**Definice 4.2.1.** Nechť $G = (N, T, P, S)$.
Precedenční syntaktický analyzátor pracující na základě $G$ používá tyto operace pro práci se zásobníkem:

- $SHIFT(<)$ přesune řetězec $\gamma =< a,\ a \in T$ na vrchol zásobníku a přečte další symbol ze vstupu,
- $SHIFT(=)$ přesune vstupní symbol $a \in T$ na vrchol zásobníku a přečte další symbol ze vstupu,
- $REDUCE(A \to \alpha)$, kde $A \in N,\ \alpha = a\beta,\ a \in T,\ \beta \in (N \cup T)^*$ nahradí řetězec $\alpha$ za $A$ na vrcholu zásobníku.

Informace k algoritmu čerpány z [9, 12], inspirace zápisu čerpána z [6].

**Algoritmus 4.2.1** Precedenční syntaktický analyzátor

**Vstup:** Precedenční tabulka pro $G = (N, T, P, S)$, $x \in T^*$ ukončený symbolem \$
**Výstup:** Pravý rozbor $x$, pokud $x \in L(G)$; jinak chyba

```
1:  stack.push($)
2:  repeat
3:      switch Tabulka[topmost_token, input_token] :
4:          case = :
5:              SHIFT(=)
6:          case < :
7:              SHIFT(<)
8:          case > :
9:              if stack.top() = < α and r : A → α ∈ R then
10:                 REDUCE(A → α)
11:             else
12:                 chyba                ▷ neexistuje pravidlo pro řetězec na zásobníku
13:             end if
14:         case × :
15:             chyba                    ▷ tabulka detekovala chybu
16:         case ✓ :
17:             úspěch
18: until chyba or úspěch
```

Jednotlivé kroky analýzy jsou určeny precedenční tabulkou, jejíž řádek je indexován nejvrchnějším (*topmost*) terminálem na zásobníku a sloupec je indexován vstupním terminálem, případně symbolem \$. Pokud je v políčku tabulky symbol = nebo <, pak analyzátor provede operaci $SHIFT$. Je-li varianta $SHIFT(<)$, pak je nejdříve na zásobník vložen symbol <, a to před první terminál, který se na zásobníku vyskytuje – pokud je na vrcholu zásobníku neterminál, přeskočí se. Slouží k rozpoznání řetězce, který má být v rámci redukce na zásobníku nahrazen. Poté se vloží na zásobník i vstupní symbol a přečte se nový ze vstupního řetězce. Pokud je v tabulce na políčku >, pak proběhne operace $REDUCE$, při které:

1. najde na zásobníku řetězec $\alpha$, který má být zredukován (přepsán) řetězcem novým,
2. určí pravidlo, u kterého se pravá strana rovná řetězci $\alpha$,
3. nahradí řetězec $\alpha$ levou stranou určeného pravidla.

Když tabulka vrátí symbol $\times$, nastává chyba. Při symbolu ✓analýza končí úspěšně.

Na následujícím příkladu z [9] je ukázán princip fungování algoritmu 4.2.1.

**Příklad 4.2.1.** Nechť $G = (N, T, P, S)$ je BKG a nechť množina $R$ obsahuje tato pravidla:

$$1 : C \to C \vee C$$
$$2 : C \to C \wedge C$$
$$3 : C \to (C)$$
$$4 : C \to i$$

A předpokládejme precedenční tabulku s těmito hodnotami:

| | $\wedge$ | $\vee$ | $i$ | $($ | $)$ | \$ |
|---|---|---|---|---|---|---|
| $\wedge$ | $>$ | $>$ | $<$ | $<$ | $>$ | $>$ |
| $\vee$ | $<$ | $>$ | $<$ | $<$ | $>$ | $>$ |
| $i$ | $>$ | $>$ | $\times$ | $\times$ | $>$ | $>$ |
| $($ | $<$ | $<$ | $<$ | $<$ | $=$ | $>$ |
| $)$ | $>$ | $>$ | $\times$ | $\times$ | $>$ | $>$ |
| \$ | $<$ | $<$ | $<$ | $<$ | $\times$ | ✓ |

Tabulka 4.2.1: Precedenční tabulka pro pravidla $G$.

Nechť vstupní řetězec je $i \wedge (i \vee i)\$$. **[[obrazek dvou moznych derivacnich stromu]]**

Analýza toho řetězce je znázorněna v tabulce 4.2.2.

| **Zásobník** | **Precedence** | **Vstup** | **Akce** |
|---|---|---|---|
| $\underline{\$}$ | $[\$, i] \to <$ | $\underline{i} \wedge (i \vee i)\$$ | $SHIFT(<)$ |
| $\$ < \underline{i}$ | $[i, \wedge] \to >$ | $\underline{\wedge}(i \vee i)\$$ | $REDUCE(C \to i)$ |
| $\underline{\$}C$ | $[\$, \wedge] \to <$ | $\underline{\wedge}(i \vee i)\$$ | $SHIFT(<)$ |
| $\$ < C\underline{\wedge}$ | $[\wedge, (] \to <$ | $\underline{(}i \vee i)\$$ | $SHIFT(<)$ |
| $\$ < C\wedge < \underline{(}$ | $[(, i] \to <$ | $\underline{i} \vee i)\$$ | $SHIFT(<)$ |
| $\$ < C\wedge < (< \underline{i}$ | $[i, \vee] \to >$ | $\underline{\vee} i)\$$ | $REDUCE(C \to i)$ |
| $\$ < C\wedge < \underline{(}C$ | $[(, \vee] \to <$ | $\underline{\vee} i)\$$ | $SHIFT(<)$ |
| $\$ < C\wedge < (< C\underline{\vee}$ | $[\vee, i] \to <$ | $\underline{i})\$$ | $SHIFT(<)$ |
| $\$ < C\wedge < (< C\vee < \underline{i}$ | $[i, )] \to >$ | $\underline{)}\$$ | $REDUCE(C \to i)$ |
| $\$ < C\wedge < (< C\underline{\vee}C$ | $[\vee, )] \to >$ | $\underline{)}\$$ | $REDUCE(C \to C \vee C)$ |
| $\$ < C\wedge < \underline{(}C$ | $[(, )] \to =$ | $\underline{)}\$$ | $SHIFT(=)$ |
| $\$ < C\wedge < \underline{(}C\underline{)}$ | $[), \$] \to >$ | $\underline{\$}$ | $REDUCE(C \to (C))$ |
| $\$ < C\underline{\wedge}C$ | $[\wedge, \$] \to >$ | $\underline{\$}$ | $REDUCE(C \to C \wedge C)$ |
| $\underline{\$}C$ | $[\$, \$] \to$ ✓ | $\underline{\$}$ | úspěch |

Tabulka 4.2.2: Demonstrace algoritmu 4.2.1.

**[[obrazek derivacniho stromu]]**

### Konstrukce precedenční tabulky

Doposud nebylo zmíněno, co znamenají položky precedenční tabulky pro analýzu výrazů. Znaménka $<$, a $>$ se interpretují jako priorita operátotů. Mějme dva operátory, například $\odot$ a $\triangle$. Pokud $\odot < \triangle$, pak $\triangle$ má vyšší prioritu, a v algoritmu to znamená, že pravá strana pravidla obsahující operátor $\triangle$ bude redukována dříve než pravá strana obsahující operátor $\odot$ a naopak. Pravidla pro tvorbu tabulky jsou pak následující [9, 12]:

1. Pokud operátor $\triangle$ má vyšší matematickou prioritu než operátor $\odot$, pak $\triangle > \odot$ a $\odot < \triangle$.
2. Pokud operátory $\triangle$ a $\odot$ jsou stejné matematické priority a levě asociativní, pak $\triangle < \odot$ a $\odot < \triangle$. Pro pravě asociativní operátory platí $\triangle > \odot$ a $\odot > \triangle$.
3. Pokud $a \in T$ může předcházet operandu $i$, pak $a < i$.
4. Pokud $a \in T$ může následovat za operandem $i$, pak $i > a$.
5. Závorky:
   (a) Pro jeden pár závorek platí $(=)$.
   (b) Pro $a \in T \setminus \{), \$\}$ platí $(< a$.
   (c) Pro $a \in T \setminus \{(, \$\}$ platí $a >)$.
   (d) Pokud $a \in T$ může předcházet $($, pak $a < ($.
   (e) Pokud $a \in T$ může následovat za $)$, pak $) > a$.
6. Nechť $\odot$ je libovolný operátor. Pak platí $\$ < \odot$ a $\odot > \$$. Dále platí $\$ \checkmark \$$.
7. Zbytek tabulky je vyplněn symbolem $\times$.

Precedenční analýza nedokáže pracovat s $\varepsilon$-pravidly a pravidly se stejnou pravou stranou pro jiné neterminály. V praxi se tedy většinou používá výhradně pro analýzu výrazů. Často se kombinuje s analýzou shora dolů (ať už prediktivní analýzou či rekurzívním sestupem), kdy analýza shora dolů kontroluje vstupní řetězec mimo výrazy a pro výrazy předává řízení analýze precedenční.

# Kapitola 5

# Implementace syntaktického analyzátoru pro jazyk Koubp

## 5.1 Přijímaný jazyk

Jazyk *Koubp* je založený na jazyce IFJ22, který je podmnožinou jazyka PHP 8, jenž byl specifikován v rámci zadání projektu do předmětu Formální jazyky a překladače v akademickém roce 2022/2023. **[[je mozne citovat zadani projektu? pokud ano, jak se k nemu dostat?]]**

### Obecné vlastnosti

Jazyk Koubp je strukturovaný jazyk. Podporuje deklaraci funkcí a proměnných, které při deklaraci musí být explicitně definovány. Hlavní tělo programu se skládá z prolínání sekvence příkazů a definic funkcí, přičemž definice funkcí nemohou být vnořené, ale mohou se vzájemně rekurzívně volat. Vstupním bodem programu není funkce `main()`, jak lze nalézt například u jazyka C [3], analýza probíhá od začátku souboru. Není podporováno slučování souborů se zdrojovým kódem pomocí příkazů `import`, `include` a podobných a tím vytvoření jediného modulu, který by bylo možné zkompilovat. Ve zdrojovém souboru a uživatelem definovaných funkcích může být větvení, iterace a další běžné konstrukce. Veškeré proměnné jsou lokální, i v rámci hlavního těla programu. Proměnné mají rozsah platnosti v *bloku kódu*, což je sekvence příkazů ve složených závorkách `{}`. Tyto bloky kódu se mohou libovolně zanořovat. V jazyce Koubp nejsou důležité *bílé mezery (whitespaces)*. Klíčová slova jazyka jsou

`if, while, for, return, elseif, else, function, if, float, string, bool.`

Identifikátory proměnných jsou posloupnosti alfanumerických znaků a znak _; identifikátor nesmí začínat číslicí. Regulárním výrazem zapsáno jako:

`[a-zA-Z_][a-zA-Z0-9_]*`

Jazyk Koubp podporuje čtyři datové typy, a to `int, float, bool, string`. Jejich literály jsou definovány následovně:

- Celočíselné literály jsou neprázdná posloupnost číslic, regulární výraz je

  `[0-9]+`

- Literály pro desetinná čísla, respektive čísla s plovoucí desetinnou čárkou, jsou neprázdná posloupnost číslic následovaná tečkou, za kterou je další neprázdná posloupnost číslic.

```
[0-9]+.[0-9]+
```

- Literály typu `bool` jsou `true` a `false`.
- Řetězcové literály jsou posloupností (i prázdnou) znaků uzavřenou v uvozovkách. Pro zapsání speciálních znaků je možnost použít *escape sekvence*, které začínají zpětným lomítkem.

```
\"(\\.|[^\\"])*\"
```

Jsou také podporovány řádkové a blokové komentáře. Řádkový komentář je sekvence znaků počínající dvojitým lomítkem `//`, končí na konci aktuálního řádku. Blokový komentář může být na více řádků počínaje znaky `/*`, konče znaky `*/`.

### Deklarace a definice funkcí

Jak bylo zmíněno v úvodu této kapitoly, deklarované funkce musí být rovnou definovány. Jazyk Koubp nepodporuje *přetěžování (overloading)*[1] funkcí. Definice funkce se skládá z hlavičky za klíčovým slovem `function` a těla funkce. V hlavičce se definuje název funkce, návratový typ a počet parametrů a jejich datové typy. Schematicky vypadá následovně:

```
function <jméno> ( <seznam parametrů> ) : <datový typ> { <tělo funkce> }
```

Dvojtečka před datovým typem musí být uvedena vždy, a to i v případě, že funkce nemá návratovou hodnotu (takzvaně `void` funkce). V tomto případě za dvojtečkou ihned následuje levá složená závorka `{`. Parametry funkce jsou oddělené čárkami a jejich struktura je:

```
<datový typ> <jméno proměnné>
```

Tělo funkce je pak neprázdná sekvence příkazů. Příklad definice funkce je:

**[[doplnit definici funkce treba faktorial nebo neco jednoduchyho]]**

### Příkazy

Jazyk Koubp podoruje obvyklé příkazy a konstrukce programovacích jazyků.

- *Přiřazení* – stejně, jako u definic funkcí, deklarace není možná. Hodnota se může přiřadit do již definované proměnné nebo při vytvoření nové proměnné, která předtím definovaná nebyla.

```
<datový typ> <jméno proměnné> = <hodnota>
             <jméno proměnné> = <hodnota>
```

- *Větvení*
- *Cyklus while*
- *Cyklus for*
- *Volání funkcí*

[1] **[[vlozit odkaz na definici overloadingu]]**

### Výrazy

- Operátory
- Priorita operátorů
- Volání funkcí

## 5.2 Gramatický systém definující syntax jazyka Koubp

**[[samotne gramatiky patri do prilohy nebo je mam hodit sem?]]**
**[[základní popis použitého GS, jak je spojený s CDGS a potažmo PCGS]]**
**[[popsat možný deadlock mezi neterminály codeblock a statement]]**
**[[indexace neterminálů]]**
**[[ll tabulka, která obsahuje uspořádané dvojice]]**

## 5.3 Návrh řešení syntaktického analyzátoru

Implementace syntaktické analýzy je silně objektově orientována, veškeré datové struktury jsou reprezentovány třídami. Třídy reprezentující neterminály a terminály mají společnou nadtřídu, díky čemuž je možné je ukládat do jednoho zásobníku za pomoci šablon. Jednu nadtřídu mají také gramatiky, jednotlivé instance jsou poté konstruovány pomocí tovární metody. Analyzátory jsou také reprezentovány třídami, společnou nadtřídu ale nemají.

### Práce s gramatikami

Gramatiky jsou rozděleny tak, aby každá z nich tvořila ucelenou část jazyka Koubp. Mimo čtvrtou gramatiku jsou všechny využívány pouze pro prediktivní analýzu. Ačkoliv téma čtvrté gramatiky je analýza výrazů, je využívána jak precedenčním, tak prediktivním analyzátorem. Důvod je zvolený postup prediktivní analýzy volání funkcí – o samotné výrazy bez volání funkcí, tedy pouze s konstantami, proměnnými a podobně, se postará analýza precedenční.

Obrázek 5.1: Schéma analýzy

**[[Tady bude obrázek znázorňující oba dva parsery a šest gramatik a který s čím pracuje.]]**

### Předávání řízení prediktivní a precedenční analýzy

Při nutnosti předání řízení druhému analyzátoru je na místě vytvořena nová instance a okamžité zavolání metody `Parse()`.

Jedním z nich je neterminál `Expression` na vrcholu zásobníku pro prediktivní analýzu. V tomto případě je zřejmé, že musí být předáno řízení precedenční analýze, aby mohla

Obrázek 5.2: Výtažek z kódu

**[[tady bude vytazek z kodu pro prepinani analyzatoru.]]**

být provedena analýza výrazu. Pokud precedenční analýza narazí na token `tFuncName`, reprezentující počátek volání funkce, musí opět předat řízení zpět prediktivní analýze.

Obrázek 5.3: Obrazek volani funkce a stridani analyzatoru
**[[Tady bude obrazek, kde bude napsany jednoduchy kod volani funkce, jeho reprezentace v tokenech a navic vlozene pomocne tokeny, ktere parsery vyuzivaji. Bude tam zobrazene, v jakych mistech se provadi zmena analyz.]]**

## 5.4 Lexikální analýza a nástroj Flex

Pro usnadnění práce byl lexikální analyzátor automaticky vygenerován nástrojem *Flex*[2] ze souboru s lexémy popsanými regulárními výrazy. S každým úspěšně analyzovaným lexémem následuje vložení tokenu do vstupní pásky pro syntaktickou analýzu. Znamená to tedy, že se nejdříve v celém zdrojovém souboru ověří lexikální správnost, až poté správnost syntaktická. Při konstrukci tokenu se inicializuje i jeho datová část, do kterého patří datový typ a samotná data. Data jsou inicializována i pro tokeny, které žádná data v programu nereprezentují –

## 5.5 Abstraktní syntaktický strom

[2]Manuál k programu k dispozici na https://westes.github.io/flex/manual/.

# Kapitola 6

# Testování

# Kapitola 7

# Závěr

# Literatura


[1] Harag, M. *Paralelní gramatické systémy: teorie, implementace a aplikace*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna, A.

[2] Hopcroft, J. E., Motwani, R. a Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. 3. vyd. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321455363.

[3] International Organization for Standardization (ISO). *ISO/IEC 9899:2018, Programming languages - C*. International Organization for Standardization (ISO), 2018. Dostupné z: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

[4] Kari, L. a Lutfiyya, H. CHAPTER 2 PARALLEL COMMUNICATING GRAMMAR SYSTEMS Bringing PC Grammar Systems Closer to Hoare ' s CSP ' s 1. In:. 2011. Dostupné z: https://api.semanticscholar.org/CorpusID:2392372.

[5] Kožár, T. a Meduna, A. *Automata: Theory, Trends, And Applications*. 1. vyd. World Scientific Publishing Co Pte Ltd, 2023. 1–418 s. ISBN 978-981-1278-12-9. Dostupné z: https://www.fit.vut.cz/research/publication/13114.

[6] Kunda, M. *Systémy syntaktických analyzátorů*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna, A.

[7] Meduna, A. *Formal Languages and Computation: Models and Their Applications*. Taylor & Francis, 2014. ISBN 9781466513457.

[8] Meduna, A. *Deep Pushdown Automata* [online]. 2007. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:rgd:slides:deep_pda.pdf.

[9] Meduna, A. *Elements of Compiler Design*. 1. vyd. Taylor & Francis Informa plc, 2008. 304 s. Taylor and Francis. ISBN 978-1-4200-6323-3. Dostupné z: https://www.fit.vut.cz/research/publication/8538.

[10] Meduna, A. a Lukáš, R. *Modely bezkontextových jazyků* [online]. 2017. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj06-cz.pdf.

[11] Meduna, A. a Lukáš, R. *Syntaktická analýza shora dolů* [online]. 2017. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj07-cz.pdf.

[12] Meduna, A. a Lukáš, R. *Syntaktická analýza zdola nahoru* [online]. 2017. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj08-cz.pdf.

[13] Meduna, A. a Zemek, P. *Regulated Grammars and Automata*. 1. vyd. New York, NY: Springer Science+Business Media, 2014. 694 s. ISBN 978-1-4939-0368-9, 978-1-4939-0369-6, 978-1-4939-4316-6.

[14] Rozenberg, G. a Salomaa, A., ed. *Handbook of Formal Languages: Linear Modeling: Background and Application*. 1. vyd. Berlin Heidelberg: Springer, 1997. ISBN 978-3-642-08230-6.

[15] Rozenberg, G. a Salomaa, A. *Handbook of Formal Languages: Volume 1. Word, Language, Grammar*. Springer, 1997. Handbook of Formal Languages. ISBN 9783540604204.

[16] Techet, J., Masopust, T. a Meduna, A. *Cooperating Distributed Grammar Systems* [online]. 2007. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:09-cdgspres.pdf.

[17] Techet, J., Masopust, T. a Meduna, A. *Parallel Communicating Grammar Systems* [online]. 2007. Ústav Informačních Systémů, FIT VUT v Brně. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:10-pcgspres.pdf.

[18] Vaszil, G. Various communications in PC grammar systems. *Acta Cybernetica*. 1. vyd. Szeged, HUN: Institute of Informatics, University of Szeged. květen 1998, sv. 13, č. 2, s. 173–196. ISSN 0324-721X.

[19] Češka, M., Vojnar, T., Rogalewicz, A. a Smrčka, A. *Teoretická informatika: Studijní opora* [online]. Srpen 2020. Ústav Inteligentních Systémů, FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf.