VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

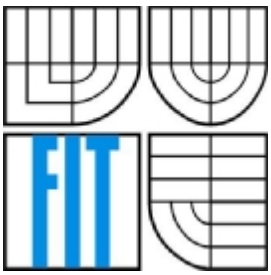
MODIFIKOVANÁ SYNTAKTICKÁ ANALÝZA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ZBYNĚK SOPUCH
AUTHOR

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MODIFIKOVANÁ SYNTAKTICKÁ ANALÝZA

MODIFICATION OF SYNTAX ANALYSIS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ZBYNĚK SOPUCH

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. RNDr. Alexandr Meduna CSc.

BRNO 2009

Abstrakt

Úkolem této práce bylo navrhnout modifikace syntaktické analýzy a zhodnotit je. Celkem jsou zde tři různé modifikace syntaktické analýze zdola nahoru, první z nich modifikuje algoritmus Cocke-Younger-Kasami, druhá využívá EOL systém a třetí systém ETOL.

Abstract

This project deals with modification of syntax analysis. I realize three modifications of bottom up algorithm. First way is modification of Cocke-Younger-Kasami algorithm, second is based on EOL system and third is based on ETOL system.

Klíčová slova

Syntaktická analýza, CYK, EOL, ETOL

Keywords

Syntax Analysis, CYK, EOL, ETOL

Citace

Sopuch Zbyněk: Modifikovaná syntaktická analýza, bakalářská práce, Brno, FIT VUT v Brně, 2009

Modifikovaná syntaktická analýza

Prohlášení

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

……
Zbyněk Sopuch
1. května 2009

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce prof. RNDr. Alexandru Medunovi CSc., za jeho odborné vedení, podporu a poskytnutou literaturu.

Obsah

1 Úvod

Syntaktická analýza je proces, během kterého zjišťujeme, zda vstupní posloupnost symbolů splňuje určitá pravidla definovaná gramatikou. Splňuje-li tyto pravidla, je zároveň rozpoznána i struktura posloupnosti a považujeme ji za syntakticky správnou, v opačném případě je zamítnuta a většinou lze identifikovat místo chyby.

Jde tedy o stěžejní proces mnohých systémů určených k analýze struktury dat, jako jsou například různé interprety, překladače, verifikátory syntaxe textu apod.

Tato analýza může být prováděna různými způsoby, které se mohou lišit jak ve směru analýzy (shora dolů, zdola nahoru), tak gramatikou a jazykem jí definovanou. Ovlivňují nejen mocnost algoritmu (jaké druhy řetězců je schopen analyzovat), ale i jeho složitost a účel použití.

Právě na toto téma, schopnosti analýzy, je zaměřena moje práce. Nejprve se podíváme, zda opravdu směr analýzy tolik ovlivňuje možnosti algoritmu a pokusíme se to vyzkoušet na algoritmu pro bezkontextovou gramatiku Cocke-Younger-Kasami (CYK). Tento algoritmus pracuje směrem zdola nahoru a naším cílem bude ho přeformulovat tak, aby byl schopen pracovat směrem shora dolů.

V druhé části práce se přesuneme do problematiky různých mocností gramatik a pokusíme se pomocí novějších technik – speciálně EOL a ETOL systémů – zvýšit mocnost analýzy bezkontextové gramatiky zdola nahoru, aby byla schopna analyzovat i posloupnosti, na které původní algoritmus nestačí.

2 Syntaktická analýza

Syntaktická analýza je proces kontroly, zda vstupní řetězec splňuje určité syntaktické podmínky, tzn., je-li větou zvoleného jazyka. Prvky řetězce budeme v analýze nazývat terminály a rozhodnutí o syntaktické správnosti se provádí na základě gramatických pravidel. Existuje velké množství různých postupů analýzy odlišných nejen tvarem pravidel, ale především její mocností – jakou třídu jazyků jsou schopny definovat.

Vstup do syntaktické analýzy je tedy tvořen posloupností terminálů, logickým výstupem je derivační strom příslušející k této posloupnosti (sestavený z pravidel gramatiky). Pokud se derivační strom podaří nalézt, posloupnost je přijata (řetězec patří do jazyka generovaného gramatikou), pokud ne, je posloupnost zamítnuta. Někdy nám stačí pouhé rozhodnutí o přijetí / zamítnutí řetězce (tak to bude i v této bakalářské práci), jindy – například v překladačích – nás zajímá i tvar stromu, který využíváme při další analýze a při generování příkazů cílového jazyka.

2.1 Gramatiky a jazyky

Abychom si mohli definovat stěžejní prvky našeho problému, kterými jsou jazyky a jejich gramatiky, musíme si nejprve říci, co je to abeceda a řetězec jí tvořený. Více informací je ve zdrojích [1], [4].

2.1.1 Abeceda

Definice 1: Nechť Σ je abeceda. Poté platí, že Σ je konečná, neprázdná množina elementů, které nazýváme symboly.

2.1.2 Řetězec

Řetězec, někdy označován jako věta, je posloupnost symbolů z abecedy.

Definice 2: Nechť Σ je abeceda a ε je prázdný řetězec nad abecedou Σ. ax je řetězec nad Σ, pokud x je řetězec nad Σ a zároveň a leží v Σ.

Délka řetězce x se označuje $|x|$ a udává počet symbol v řetězci. Musí však platit: $|x| >= 0$.

2.1.3 Gramatika

Gramatika hraje v syntaktické analýze důležitou roli. Na jedné straně stojí řetězec, který budeme analyzovat, a na straně druhé stojí právě gramatika jako nástroj použitý k analýze.

Definice 3: Gramatika je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \emptyset$
- P je konečná množina pravidel $\alpha \rightarrow \beta$, kde $\alpha \in (N \cup T)^*N(N \cup T)^*$, $\beta \in (N \cup T)^*$
- S je počáteční neterminál, přičemž platí, že $S \in N$

Terminál je symbol, který se může nacházet v příchozí posloupnosti symbolů, jde tedy o abecedu vstupního řetězce. Naopak neterminál je symbol z dalších úrovní derivačního stromu a může se nacházet pouze v pravidlech, ale již ne ve vstupním řetězci. Pokud nebudeme potřebovat rozlišit symbol jako terminál, neterminál, budeme ho souhrnně nazývat token.

Strana pravidla β se nazývá pravou stranou a určuje posloupnost, která se má přepsat. Strana α je levá strana a určuje, na co se má posloupnost přepsat.

2.1.4 Jazyk generovaný gramatikou

K určení jazyka generovaného gramatikou musíme nejprve pochopit postup aplikace pravidel. Tento proces budeme nazývat derivační krok.

Definice 4: Máme gramatiku G = (N, T, P, S). Nechť $u, v \in (N \cup T)^*$ a $p = \alpha \rightarrow \beta \in P$. Potom $u\alpha v$ přímo derivuje $u\beta v$ za použití pravidla p v G.

Sekvenci 0 až n derivačních kroků (kde $n > 0$) budeme značit $u \Rightarrow^* v$; jde o tranzitivní a reflexivní uzávěr relace přímé derivace a znamená, že jsme se z posloupnosti u dostali pomocí n aplikací pravidel k posloupnosti v.

Nyní již můžeme definovat jazyk generovaný gramatikou.

Definice 5: Máme gramatiku G = (N, T, P, S). Jazyk generovaný gramatikou G, značíme L(G), je definován $L(G) = \{u: u \in T^*, S \Rightarrow^* u\}$.

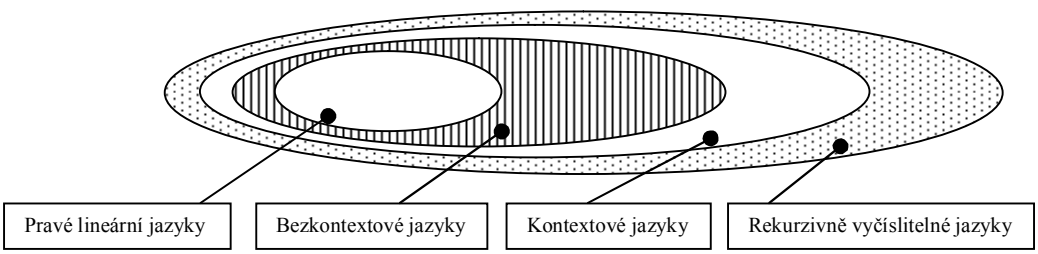
2.1.5 Chomského klasifikace gramatik

V *tabulce 1* je uvedena klasifikace gramatik a jazyků jimi generovaných od těch nejobecnějších, po ty nejvíce specifické.

Gramatiky	Generované jazyky	Tvar pravidel gramatiky
Neomezené	Rekurzivně vyčíslitelné	$A \in (N \cup T)^*N(N \cup T)^*, x \in (N \cup T)^*$
Kontextové	Kontextové	$A \in (N \cup T)^*N(N \cup T)^*, x \in (N \cup T)^*, \lvert a\rvert < \lvert x\rvert$
Bezkontextové	Bezkontextové	$A \in N, x \in (N \cup T)^*$
Pravé lineární	Regulární	$A \in N, x \in T^* \cup T^*N$

Tabulka 1: Chomského klasifikace gramatik

Schematicky si můžeme množinu jazyků představit tak, jak jsou rozděleny na *obrázku 1*.



Obrázek 1: Schéma klasifikace jazyků

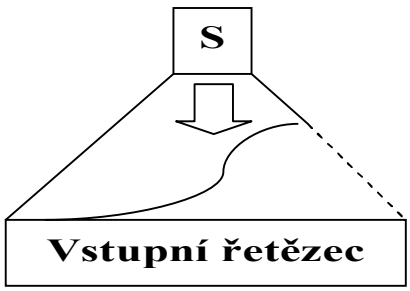
2.2 Základní typy syntaktické analýzy

Rozeznáváme dva základní typy syntaktické analýzy podle směru aplikace pravidel: shora dolů a zdola nahoru.

2.2.1 Syntaktická analýza shora dolů

Syntaktický analyzátor shora dolů vytváří derivační strom řetězce jazyka od počátečního neterminálu (kořene stromu) a postupně doplňuje aplikací pravidel hrany a uzly směrem dolů do listů. Proces je úspěšný (řetězec vyhovuje gramatice) v případě, že listy tvoří identickou posloupnost s posloupností symbolů v řetězci. U tohoto postupu je potřeba se dobře vypořádat s determiničností gramatiky, aby při její aplikaci byl výběr pravidla jasně definován, neboť symboly analyzovaného řetězce leží až v nejnižším levelu stromu = v listech. Jako možné řešení se používá LL-gramatika, u které vždy existuje maximálně jedno pravidlo vyhovující aktuálnímu stavu.

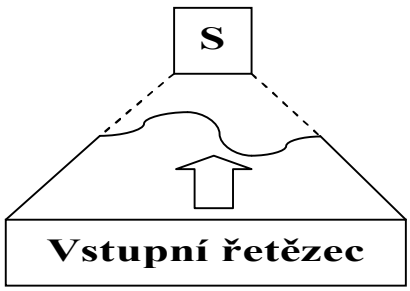
Poznámka: Princip použití LL-gramatiky znamená, že se vždy provádějí pouze nejlevější chybějící derivace a gramatika je sestavena tak, že výběr pravidel bude při tomto postupu deterministický.



Obrázek 2: Metoda shora dolů

2.2.2 Syntaktická analýza zdola nahoru

U syntaktického analyzátoru zdola nahoru postupuje přesně v opačném směru než u předchozího typu. Pravidla gramatiky se začínají aplikovat od symbolů řetězce, derivační strom je tedy tvořen postupnou aplikací pravidel od listů ke kořeni. Proces je úspěšný v případě, že jsme našli kořen stromu totožný s počátečním neterminálem S. Problémy, se kterými se zde musíme vypořádat, jsou pravidla se stejnou pravou stranou (nevíme, které použít) a možnost vytvořit více různých stromů vedoucích ke kořeni (tento problém se dá řešit například prioritami).



Obrázek 3: Metoda zdola nahoru

3 Modifikace CYK algoritmu

Jeden z faktorů ovlivňujících použití algoritmu, je směr, kterým pracuje. K analýze některých typů řetězců, například výrazů, je lepší přístup zdola nahoru, k jiným typům řetězců, například v případě analýzy struktury dokumentu, je jednodušší použít směr shora dolů. Teoreticky lze oba přístupy aplikovat v obou případech, avšak zvolení nesprávného přístupu by mohlo vést ke zbytečnému zesložitění problému.

Podle zvoleného přístupu si vybíráme i použitý algoritmus. Tyto algoritmy jsou navrženy tak, aby fungovaly jedním směrem a maximálně využívaly výhod tohoto přístupu. Naším cílem je na příkladu ukázat, že směr analýzy a algoritmus nejsou striktně provázány, ale že lze po určitých úpravách směr algoritmu otočit.

Otočený algoritmus musí minimálně splňovat zachování stejných vstupních a výstupních podmínek. Navíc by měly být v průběhu procesu tvořeny i podobné struktury, ke kterým se však dostaneme opačným směrem.

K tomuto účelu jsme si vybrali algoritmus Cocke-Younger-Kasami (CYK) pro bezkontextovou gramatiku, který pracuje směrem zdola nahoru.

3.1 CYK algoritmus

CYK algoritmus analyzuje syntaktickou strukturu řetězce a to směrem zdola nahoru. Je určený pro analýzu bezkontextových jazyků a využívá bezkontextové gramatiky ve speciální Chomského normální formě. Tento algoritmus je oblíbený především pro jeho jednoduchost a snadnou implementovatelnost. Více informací lze nalézt ve zdroji [2].

3.1.1 Chomského normální forma gramatiky

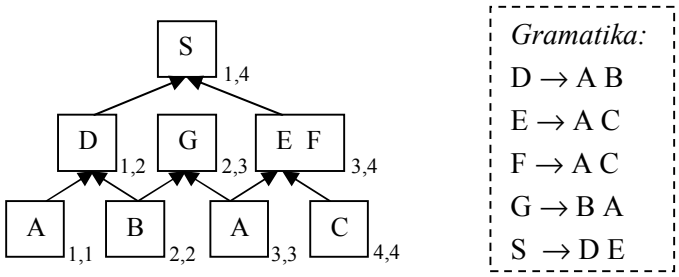
Podle zvláštního tvaru pravidel můžeme rozpoznat určité normální formy gramatik. Nás bude zajímat speciálně Chomského normální forma, kde mohou pravidla nabývat pouze tyto tvary:

- $A \rightarrow BC$, kde A, B, C jsou neterminály
- $A \rightarrow a$, kde *a* je terminál
- Je-li v gramatice i prázdný řetězec ε, tak $S \rightarrow \varepsilon$, přičemž S je počáteční neterminál

Pro každou bezkontextovou gramatiku existuje ekvivalentní gramatika v Chomského normální formě.

3.1.2 Popis algoritmu

Zjednodušeně pracuje algoritmus tak, že vytváří množiny neterminálů od symbolů řetězce nahoru a vždy do nich vloží všechny levé strany těch pravidel, pro které lze pravou stranu aplikovat na příslušnou pozici dvou symbolů o úroveň níže (můžeme vidět na *obrázku 4*).



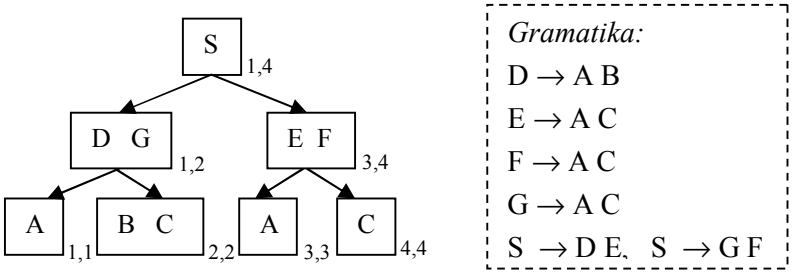
Obrázek 4: Algoritmus CYK

Formální popis algoritmu CYK pro vstupní řetězec x a gramatiku G = (T, N, P, S):

- Vytvořme množiny CYK[i,j] = ∅, pro $1 \leq i \leq j \leq n$, kde $n = |x|$
- Pro všechna i od 1 do n, kde existuje, $X \rightarrow a_i \in$ P, $a_i \in$ N, $X \in$ T, vložíme X do CYK[i,i]
- Opakujeme, dokud jsou změny
  - → Pokud $B \in$ CYK[i,j], $C \in$ CYK[j+1,k], $A \rightarrow BC \in$ P, A i B i $C \in$ N, vložíme A do CYK[i,k]
- Pokud S ∈ CYK[1,n], tak $x \in$ L(G)

3.2 Varianta 1 – prosté otočení algoritmu

Nejprve logiku procesu pouze otočíme. Jak jsem zmínil dříve, algoritmy jsou navrženy tak, aby využívaly výhody svého směru, je proto velmi pravděpodobné, že po otočení nebudou fungovat správně. Pro nás to však bude model, na kterém se pokusíme identifikovat hlavní problémy. Ilustračně je postup předveden na *obrázku 5*.



Obrázek 5: Otočený proces algoritmus CYK

Formální popis otočeného algoritmu shora dolů pro vstupní řetězec x a gramatiku G = (T, N, P, S):

- Vytvořme množinu CYK[i,j] = [S], pro $1 \leq i \leq j \leq n$, kde $n = |x|$
- Opakujeme, dokud je změna:
  - → Sestrojíme množiny neterminálů CYK[i,j], CYK[j+1,k], $1 \leq i \leq j \leq k \leq n$, tak, že $A \in$ CYK[i,j] a $B \in$ CYK[j+1,k] tehdy a jen tehdy, pokud $C \in$ CYK[i,k] a $C \rightarrow AB \in$ P
- Pokud existuje $X \rightarrow a_i \in$ P pro všechna i, $1 \leq i \leq$ n, tak, aby $X \in$ CYK[i,i], tak $x \in$ L(G)

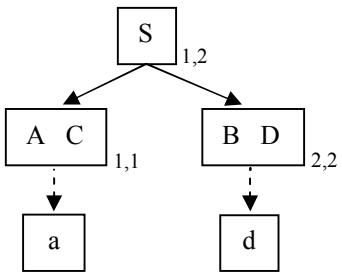
3.2.1 Problémy a poznatky

Algoritmus je konečný právě proto, že je konečný i řetězec. Každé pravidlo obsahující pouze neterminály větví strom, ale ten nemůže být rozvětvený více, než je délka řetězce, proto je konečné i větvení. Pravidla s terminály sice strom nevětví, ale mohou být použity pouze při překladu neterminálů na řetězec (žádný terminál se nemůže objevit na levé straně pravidla), k zacyklení tedy nedojde. Plyne z toho i další poznatek: k úspěšné analýze musíme sestrojit právě n množin s terminály ve tvaru CYK[i,i], pro všechna $1 \leq i \leq n$, reprezentující právě n symbolů řetězce, jinak může být řetězec rovnou odmítnut ($x \notin$ L(G)).

Během vytváření indexů množin se při nesouměrnosti stromu mohou vyskytnout nejasná místa. Tyto problémy budeme prozatím ignorovat, neboť prvotní algoritmus není v použitelném stavu. Navíc, na rozdíl od původního CYK algoritmu, zde nejsou indexy příliš důležité. Prakticky stačí, pokud budeme strom větvit i bez indexů a po dosažení tolika množin s terminály, kolik je symbolů v řetězci, můžeme proces zastavit a provést srovnání (konečnost algoritmu z odstavce výše stále platí).

Tento jednoduchý algoritmus může i bude ve velmi specifických případech fungovat. Je tu ale zásadní problém v případě, že pro některý z kroků existuje více možných pravidel (to znamená, že sestrojené množiny budou mít více než 1 prvek). Při aplikování pravidel od řetězce směrem nahoru jsme spojovali paralelní větve, naopak nyní paralelní větve vytváříme. Tento zdánlivě malý rozdíl má veliké důsledky. Pokud totiž strom větvíme, nevíme, jak budou dále větve pokračovat a z jakých pravidel byly stvořeny. Vzniknou tak správné množiny neterminálů, které však mezi sebou na stejné úrovni nemají žádné vazby. Je to problém především u závěrečného rozpoznávání řetězce, neboť právě spojování terminálů do řetězce vyžaduje vazby na poslední úrovni.

Máme-li například gramatiku G s pravidly S → *AB*, S → *CD*, *A* → *a*, *D* → *d* a řetězec *x* = „*ad*“, tak tento řetězec nepatří do jazyka L(G). Na *obrázku 6* však uvidíme, jak bude vypadat strom po provedení algoritmu.



Obrázek 6: Problémy otočeného algoritmu

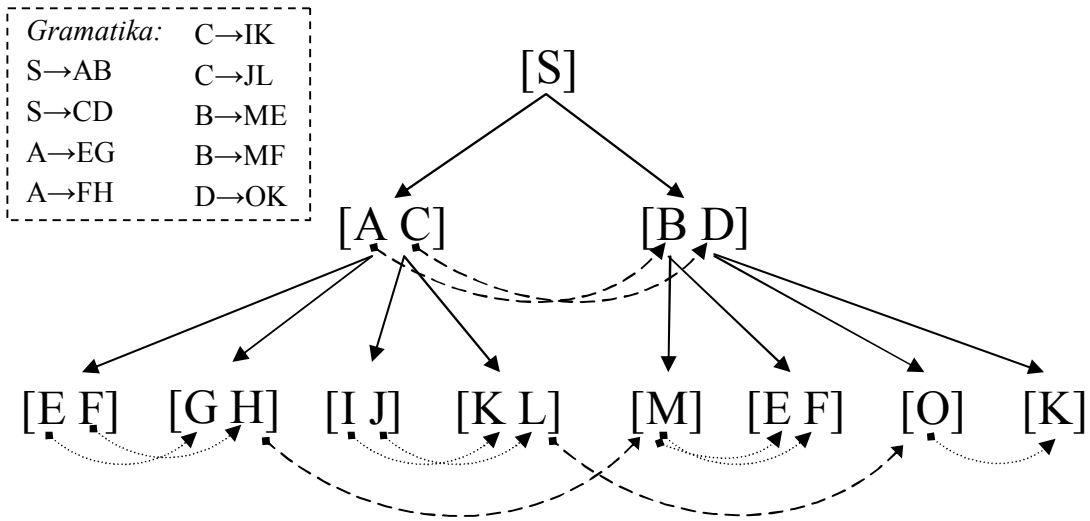
Vznikl nám strom, podle kterého řetězec „*ad*“ do jazyka patří. Na vině je ztráta vazeb v pravidle při rozdělení na paralelní větve. Máme sice množiny správně, opravdu se neterminálů *A* a *C* mají nacházet v levé množině, ale již nevíme, jestli *A* patří k *B* nebo *D*, tedy z jakého pravidla jsou.

Pokud chceme zachovat algoritmus funkční, musíme si nějakým způsobem označit nebo pamatovat, které symboly pocházejí ze stejného pravidla. Na možnosti, jak toto uskutečnit, se podíváme ve *variantě 2* (kapitola 3.3) a *variantě 3* (kapitola 3.4).

3.3 Varianta 2 – ukazatele

Jeden ze způsobů, jak neztratit záznamy o závislostech, je použití ukazatelů. Jinak je postup stejný jako ve *variantě 1* (kapitola 3.2).

Budeme si tedy s každým prvním neterminálem z pravé části pravidla ukládat i ukazatel na neterminál druhý (pravidla s neterminály jsou binární). Po hlubší analýze přijdeme na to, že nestačí si pamatovat pouze informaci, které dva tokeny ve stromě patří k sobě, ale musíme tuto závislost dědit i k množinám od nich odvozených. Proto je nezbytné v každé úrovni vytvářet nejen množiny zvlášť pro první a druhý token pravé strany pravidla, ale i zvlášť pro různé pravé strany. Lépe to vše půjde vidět na *obrázku 7*.



Obrázek 7: Vztahy pomocí ukazatelů

V původním algoritmu by byly množiny [E F] a [I J] spojeny v jednu, stejně tak množiny [G H] a [K L]. Avšak v rámci uchování vazeb musí být tyto množiny zvlášť, protože každá vznikla z pravidel s jinou pravou stranou; [E F] vzniklo z neterminálu A, [I J] z neterminálu C.

Nyní již lze rozhodnout, které tokeny patří ke kterým, neboť množiny byly pospojovány v různé varianty řetězce, kde každá množina reprezentuje jeden symbol na jedné pozici. Hledání prvního symbolu výsledné posloupnosti můžeme v uvedeném příkladu začít v množině [E F] nebo [I J]. To jsou jediné množiny, na které neukazují žádné nové ukazatele. Ukazatele z nich vycházející nás dovedou k dalším množinám, ve kterých hledáme tokeny pro následující pozice.

Vidíme však, že je to nepřehledný systém generující větší množství množin než původní CYK algoritmus (a ve kterém by nastal ještě větší zmatek, pokud by nebyly oba tokeny z pravé strany pravidla na stejné úrovni, ale některý z nich by měl více generací potomků než druhý).

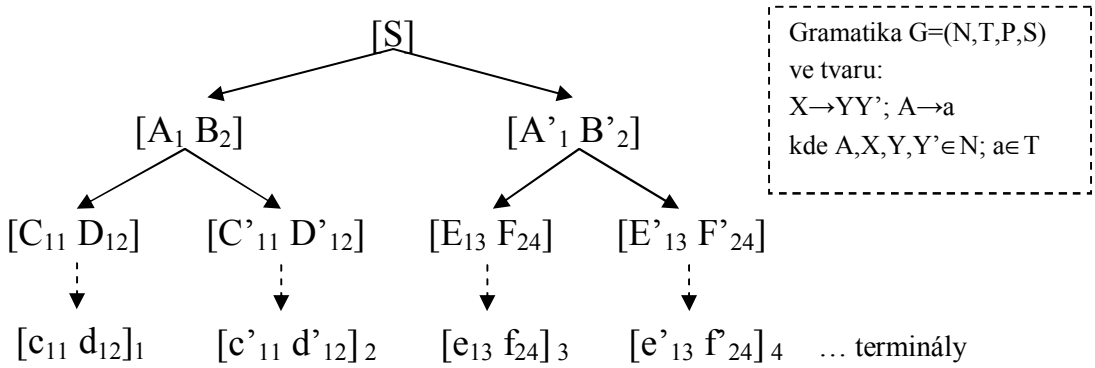
3.4 Varianta 3 – indexy tokenů

Při klasické analýze indexujeme množiny podle toho, jakou část řetězce reprezentují. Proč ale neindexovat samotné tokeny? Během celého procesu by tak vznikal nejen strom množin, jak tomu bylo doteď, ale zároveň i jakási matematická indexová reprezentace virtuálních stromů, identifikující dědičnost a vztahy mezi tokeny.

Každý token, který vkládáme do stromu, bude mít vlastní množinu indexů. Při vložení dalších tokenů aplikací některého z pravidel zdědí tyto tokeny množinu indexů po předkovi (vkládáme tokeny z pravé strany pravidla, dědí tedy po tokenu na levé straně). Dále přidáme oběma tokenům z levé strany za zděděné indexy stejný index unikátní pro aktuální aplikaci pravidla. Tato unikátnost musí být dodržena minimálně na úrovni stromu (čímž se redukuje velikost indexů, ale zvyšuje složitost jejich generování), avšak kvůli jednoduchosti můžeme použít i indexy unikátní v celém stromě (k tomuto stačí udržovat nějaký čítač počtu aplikovaných pravidel, který bude představovat další použitý index). Z tohoto principu rozšiřování indexů jsou vyloučeny pravidla s terminály (mají tvar $A \rightarrow a$, $A \in N$, $a \in T$), které pouze indexy dědí. Jde vlastně o pouhé překlopení neterminálu na terminál, proto nedochází k žádné změně.

Jestli označíme i počáteční token S, je jenom na nás. Tento token musí být právě jeden, a proto mu bude náležet vždy pouze jedno číslo, které bychom museli dědit přes celou hierarchii stromu. Jeho očíslování postup ani výsledek nijak neovlivní, pouze si o jeden krok prodloužíme závěrečnou analýzu závislostí a celý proces se stane nepatrně paměťově náročnější. Na druhou stranu to, že máme možnost do označování zahrnout všechny tokeny bez jakýchkoli rozdílů, ukazuje na čistotu algoritmu a jeho obecnost.

Celý proces bude nejprve probíhat stejně, jako ve *variantě 1* (kapitola 3.2), změna nastane až při hledání řetězce. Jak bylo řečeno již dříve, pokud se nám nepodařilo najít právě *n* množin s terminály, reprezentující právě *n* symbolů řetězce, můžeme řetězec rovnou zamítnout ($x \notin L(G)$).

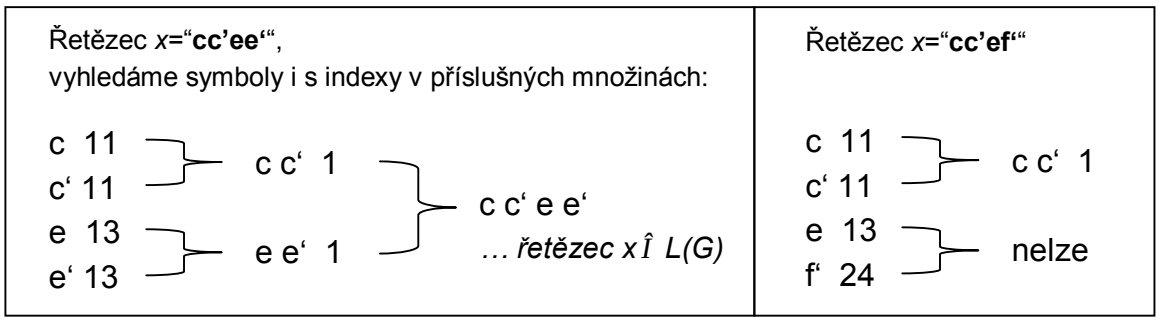


Obrázek 8: Indexace tokenů ve stromě

Obrázek 8 nám ilustruje postup tvoření stromu až po konečné množiny s terminály. Nyní musíme z těchto množin vybrat symboly řetězce i s jejich indexy, a to tak, že v množině $[\ldots]_1$ hledáme 1. symbol řetězce atd. (formálně $x_i \in [\ldots]_i$, kde x_i je i-tý symbol řetězce x a $[\ldots]_i$ i-tá množina terminálů).

Pokud jsme všechny symboly řetězce v příslušných množinách našli, postupujeme k poslední fázi, a to ke kontrole závislostí. V opačném případě, tj. některý ze symbolů nebyl nalezen, řetězec odmítneme. Dále předpokládejme, že všechny symboly máme. Postupně budeme „redukovat“ indexy u jednotlivých dvojic symbolů, které leží v řetězci vedle sebe. Jelikož byly indexy na dané úrovni stromu vždy pro pravidlo unikátní, může existovat pouze jedna dvojce se stejnými indexy. Budeme tedy opakovaně procházet všechny dvojice a u těch, které mají stejný poslední index (stejná hodnota i pozice), tento index smažeme a dvojice sloučíme v jeden symbol se stejným zbytkem indexů, jako jeden z nich (zbytek indexů by měl být shodný u obou, stačí proto vzít indexy třeba prvního; například symbol „*x*“ s indexem 1-2-1 a „*y*“ s 1-2-1 sloučíme v jeden virtuální symbol „*xy*“ s indexem 1-2). Pokud se nám podařilo takto zredukovat veškeré symboly (a jejich indexy) a vznikl nám jeden virtuální symbol obsahující celý hledaný řetězec, byl řetězec přijat a patří do jazyka. V opačném případě do jazyka nepatří.

Postup kontroly závislostí můžeme vidět v *tabulce 2*, která navazuje na *obrázek 8*.



Tabulka2: Kontrola závislostí terminálů

3.4.1 Charakteristika vlastností algoritmu

Maximální počet indexů je u tokenů v listech a je roven výšce stromu, tedy $h = \log_2 n$ (zaokrouhleno nahoru), kde n je počet symbolů v analyzovaném řetězci (h je třeba ještě opravit o ±1, záleží na zahrnutí počátečního neterminálu a množin s terminály). Z toho se odvíjí i přidaná paměťová náročnost, neboť indexy jsou jediné perzistentně uchovávané informace navíc v algoritmu.

Režii analýzy tvoří pouze kopírování indexů a na závěr jejich jednorázová redukce. Avšak i pokud pomineme tuto malou režii, bude proces pomalejší než původní CYK směrem zdola nahoru. Důvod je, že nyní vlastně generujeme všechny možné řetězce o stejné délce jako hledaný řetězec, které patří do jazyka L(G), kde G je použitá gramatika. Při analýze v původním směru jsme měli naopak prohledávaný stavový prostor omezený terminály z řetězce, které již v prvním kroku vybraly jen část pravidel. Počet uzlů ve stromu bude tedy stejný, ale počet použitých pravidel, jejichž hledání a aplikování tvoří největší část analýzy, může být až několikanásobně větší.

3.5 Varianta 4 – LL-gramatiky

Jeden z hlavních principů pro zajištění determinismu u analýzy bezkontextových jazyků směrem shora dolů, jsou LL-gramatiky. Dostáváme se však již mimo náš původní záměr, kterým bylo pouze modifikovat algoritmus, nikoliv vstupní a výstupní podmínky. Abychom mohli LL-gramatiky využít, musíme nejprve vytvořit gramatiku s novými vyhovujícími pravidly, čímž jsme ale vstupní podmínky nedodrželi.

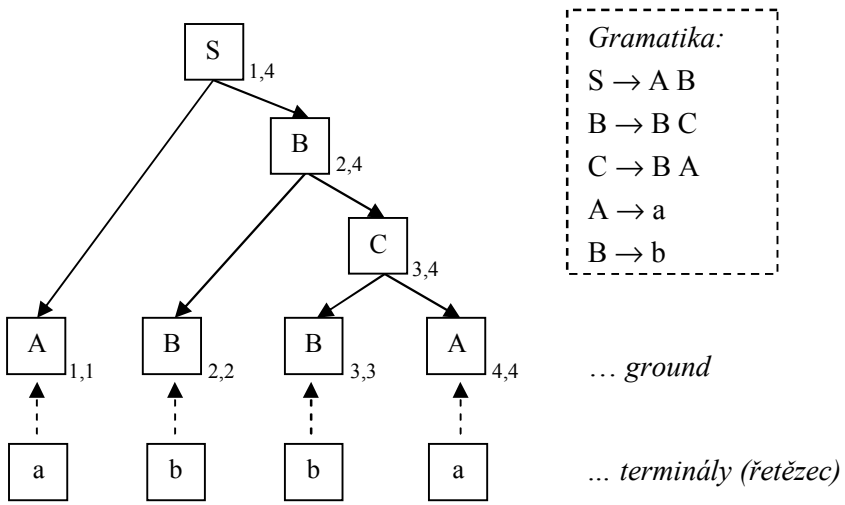
Další problém, který tuto variantu vylučuje z našeho důkazu modifikace CYK algoritmu, je změna mocnosti algoritmu. Na vině je, že tato modifikovaná gramatika nejde najít vždy (stejně jako u klasických LL-gramatik, které tvoří pouze podmnožinu gramatik bezkontextových).

Přesto je to zajímavá modifikace, kterou si alespoň naznačíme.

3.5.1 Použití LL-gramatiky v analýze

Tvar pravidel musí zůstat stejný, protože se pořád pohybujeme v analýze založené na Chomského normální formě. Vyplynulo nám však jedno nové omezení, nutné k dosažení determiničnosti při derivaci zleva: kombinace pravé strany pravidla a prvního symbolu levé strany musí být jedinečná. Formálně zapsáno pro gramatiku G = (N,T,P,S): $\alpha \in N$, $B \in (N \cup T)$, $\beta \in N^*$, pro každé $\alpha \rightarrow B\beta \in P$, platí, že v $P - \{\alpha \rightarrow B\beta\}$ neexistuje žádné pravidlo $\alpha \rightarrow B\gamma$, kde $\gamma \in N^*$.

Pokud se nám podařilo modifikovat gramatiku do požadovaného tvaru (nebo jsme ji přímo v tomto tvaru vytvořili), můžeme spustit analýzu. Tvar stromu analýzy je znázorněn na *obrázku 9*.



Obrázek 9: Použití LL-gramatiky

Postup analýzy je jednoduchý. Nejprve symboly řetězce „přeložíme“ na neterminály a tím vytvoříme úroveň „ground“. Tento krok je převzat z původní analýzy zdola nahoru. Nyní začneme postupně aplikovat pravidla od kořene (počátečního neterminálu S) směrem dolů a to tak, že první symbol levé strany pravidla musí odpovídat prvnímu nepoužitému symbolu v „ground“ (tedy u prvního pravidla jde o symbol na pozici 1,1, u druhého 2,2…). Vznikne nám tak vždy maximálně jeden volný uzel, a to právě druhý symbol z použitého pravidla. Považujme ho za nový kořen a celý proces opakujme.

Konec analýzy nastane, pokud nemáme žádné vhodné pravidlo (řetězec $x \notin L(G)$), nebo pokud nám zůstal již poslední nepoužitý symbol v „ground“. Je-li shodný s druhým symbolem pravé strany naposledy použitého pravidla, řetězec patří do jazyka gramatiky, jinak do jazyka nepatří.

Množiny není nutné indexovat, ale jak je uvedeno na *obrázku 9*, není to žádný problém. Navíc nejde o množiny v pravém slova smyslu, protože jsou vždy jednoprvkové, můžeme z nich tedy udělat přímo tokeny. Způsobuje to fakt, že máme maximálně jedno pravidlo pro každou kombinaci symbolů.

3.6 Zhodnocení modifikací

Při zpětném pohledu na jednotlivé varianty vidíme, že otočit algoritmus není až tak jednoduché, protože ztrácíme veškeré výhody, které těžil ze svého původního směru. Tyto náležitosti musíme komplikovaně doplňovat a to zvyšuje nejen režii, ale i celou složitost provedení algoritmu. Přesto jsme dospěli k závěru, že to lze. Jak modifikace *varianta 2*, tak *varianta 3* zachovala vstupy i výstupy CYK algoritmu i jeho mocnost. Navíc *varianta 3* si uchovala i částečnou podobnost s původním procesem za cenu jen mírné režie. Je tedy vidět, že směr postupu mocnost algoritmu neovlivní, ale že ovlivní jen jeho vlastnosti (rychlost, paměťová náročnost…).

Co se týče použitelnosti modifikací, pohybujeme se již za úrovní našeho vytyčeného cíle. Využití pouhého obráceného algoritmu (*varianta 1*) může být maximálně ke generování množin všech přípustných tokenů na jednotlivých pozicích všech řetězců jazyka zvolené délky (nebo naopak doplňkem k množině všech tokenů gramatiky získáme množiny nepřípustných tokenů na daných pozicích).

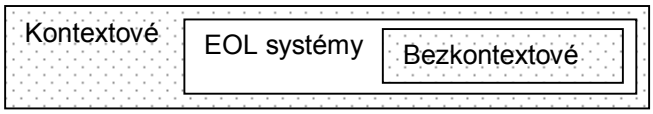
Využít postupy z *varianty 2 a 3* k analýze řetězce by již šlo, ale jelikož prohledávají větší stavový prostor (generují všechny řetězce dané délky vyhovující gramatice), jsou pomalejší a složitější, než algoritmus původní. V případě, že opravdu potřebujeme postupovat shora dolů, bychom nejspíše sáhli po jiných již používaných algoritmech.

Pro reálné nasazení se pravděpodobně hodí pouze *varianta 4* s LL-gramatikami. Jak již bylo řečeno dříve, nemůžeme ji zahrnout do našeho pokusu o otočení algoritmu, protože mění vstupní podmínky a snižuje mocnost jazyka. Přesto jako jediná modifikace neprohledává příliš široký stavový prostor, je rychlá a obsahuje i všechny ostatní výhody klasických systémů s LL-gramatikami.

4 Modifikace s využitím EOL systému

V předchozí části práce jsme se studovali, jak možnosti algoritmu ovlivní jeho směr. Nyní se podíváme na samotný algoritmus a zkusíme ho modifikovat ke zvětšení jeho analytické síly. Vyjdeme z algoritmu zdola nahoru pro bezkontextovou gramatiku, podobného CYK algoritmu popsaného v kapitole 3.1, a k jeho modifikaci využijeme takzvaný EOL systém.

Chomského klasifikace jazyků (z kapitoly 2.1.5) definuje čtyři úrovně jazyků. Do teď jsme pracovali v množině bezkontextových jazyků, abychom mohli považovat nový algoritmus za mocnější, musí umět analyzovat i některé jazyky, které do této množiny nepatří. Právě EOL systémy dokážou generovat množinu bezkontextových jazyků i část jazyků navíc, jejich zařazení můžeme vidět na *obrázku 10* [3].

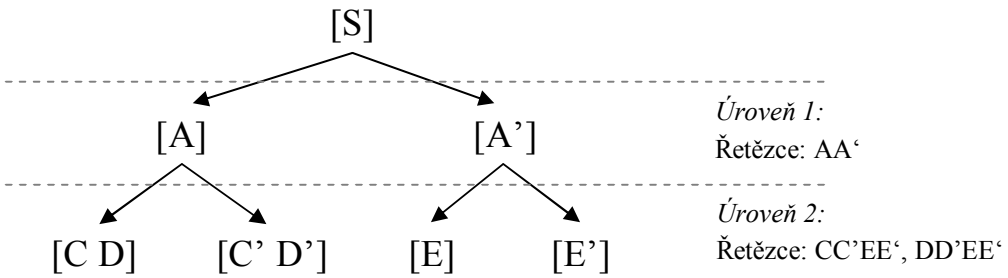


Obrázek 10: zařazení EOL systémů

4.1 EOL systém

K přesné definici EOL systému jsou vyžadovány znalosti L-systémů. Jejich definice je však mimo rozsah naší bakalářské práce. Stanovíme si tedy pouze hlavní rozdíly a pravidla analýzy a doplňující informace společně s definicí lze případně najít v disertační práci *L-Systems: New results and application* od PhD. Ing. Milana Kolky na straně 12 [3].

Stěžejní myšlenka EOL systémů je, že všechny symboly řetězce musí být vygenerovány na stejné úrovni derivačního stromu (viz. *obrázek 11*). Mimo jiné to znamená, že se v každém kroku musí pravidly přepsat všechny listové množiny.



Obrázek 11: Úrovně stromu s EOL systémem

Nyní již můžeme odhadnout omezení a dopady na gramatiku, která bude do algoritmu vstupovat. Pravidla budou ve tvaru bezkontextové gramatiky, na levé straně bude tedy právě jeden neterminál. Na straně pravé mohou být libovolné tokeny, ale na rozdíl od bezkontextové gramatiky se nesmí míchat. V jednom pravidle mohou být na této straně pouze neterminály, nebo pouze terminály, jinak bychom nemohli získat řetězec pouze z jedné úrovně stromu. Stejně tak v jedné úrovni stromu mohou být aplikovány pouze jedny z těchto dvou druhů pravidel – tedy buď s neterminály, nebo s terminály. Pokud byly aplikovány pravidla s terminály, rozvoj řetězce se ukončil.

Můžeme mít ještě jeden přístup k problémům s terminály a neterminály, který je dokonce i EOL systémům bližší. Terminály a neterminály nebudeme rozlišovat, ale sloučíme je v tokeny. Stále platí, že řetězec se musí získat z jedné úrovně stromu, tedy všechny tokeny na stejné úrovni mohou tvořit řetězec. V tom případě nebude jejich rozvoj pokračovat (nebudeme na ně aplikovat pravidla). Ovlivní to i pravidla gramatiky, neboť libovolný symbol (token) se může objevit na obou stranách pravidla.

Výběr, který přístup používat, nemá žádný větší praktický dopad. Z důvodu zachování co největší podobnosti s klasickým algoritmem použijeme k formální definici procesu přístup první.

Poznámka: Při pohledu na tento systém je vidět, že by i zde šla použít Chomského normální forma gramatiky. Forma vyhovuje podmínkám ohledně střídání terminálů / neterminálů a gramatika by nyní mohla nabýt pouze binárního tvaru. Přínosem použití takto upraveného EOL systému je možnost aplikace modifikací z kapitoly 3, přičemž by se zmenšily či úplně vymizely některé problémy, jako například problém s indexy z *varianty 1* (kapitola 3.2), neboť strom je dokonale souměrný.

4.2 Algoritmus syntaktické analýzy

Jaký je hlavní rozdíl mezi stromem EOL systému oproti klasickému derivačnímu stromu? Právě v omezení operací na jednotlivé úrovně. Nelze při jedné aplikaci pravidla zároveň vygenerovat / substituovat (záleží na směru analýzy) token například z první úrovně a token z úrovně třetí. Proto musíme v průběhu analýzy zajistit, aby při vytváření úrovně byla vidět pouze úroveň předchozí.

Jedno z možných řešení je po dokončení úrovně stromu vytvořit její dočasnou kopii a další úroveň vytvářet aplikací pravidel právě na tuto dočasnou kopii. Při definici algoritmu využijeme podobný přístup, jenom bude obrácený. Použijeme pravidla na aktuální úroveň a do pomocné úrovně budeme ukládat právě výsledky aplikace. Pracujeme-li tedy zdola nahoru, budeme provádět substituci tokenů shodných s pravou stranou pravidla a jeho levou stranu uložíme na příslušnou pozici v pomocné úrovni. Po aplikaci všech přípustných pravidel pouze překlopíme pomocnou úroveň do aktuální.

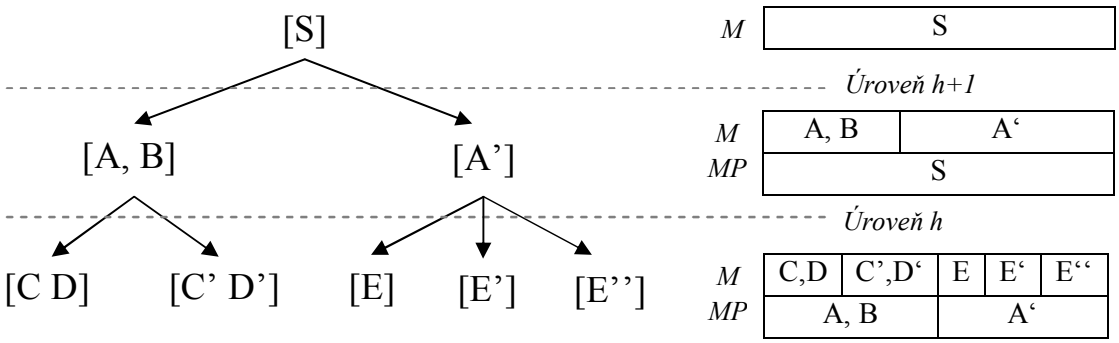
Formální zápis algoritmu s EOL systémem pro vstupní řetězec x a gramatiku G = (T, N, P, S) může vypadat takto:

- Předpokládejme, že $1 \leq i \leq j \leq k \leq n$, kde $n = |x|$
- Vytvořme množiny hlavního stromu $M[i,j] = \varnothing$
- Pro všechna i, $1 \leq i \leq$ n, provedeme $M[i,i] = x[i]$, $x[i]$ značí i-tý symbol řetězce
- Opakujeme, dokud jsou změny a zároveň $S \notin M[1,n]$
  - → Vytvořme pomocné množiny $MP[i,j] = \varnothing$
  - → Pokud najdeme posloupnost tokenů β takovou, že $\beta[g]$ je g-tý prvek posloupnosti a pro všechna g, kdy $1 \leq g \leq |\beta|-1$, platí, že $\beta[g] \in M[i,j]$ a zároveň $\beta[g+1] \in M[j+1,k]$ a zároveň $\alpha \rightarrow \beta \in P$, pak $MP[i,k] = \alpha$
  - → Pro všechna $M[i,j]$ proveďme $M[i,j] = MP[i,j]$
- Pokud $S \in M[1,n]$, tak $x \in L(G)$

Analýzu můžeme ukončit v těchto případech:

1. Dosáhneme-li kořene stromu a zároveň to bude počáteční neterminál S. Pak $x \in L(G)$
2. Existuje-li alespoň jedna množina neterminálů ve stromě, kde pro všechny neterminály neexistuje žádné aplikovatelné pravidlo, pak $x \notin L(G)$.

Názorně můžeme celý proces vidět na *obrázku 12*. Je zde vedle sebe zobrazen derivační strom s vyznačenými jednotlivými úrovněmi a pro každou úroveň vždy aktuální stav tokenové reprezentace řetězce (množiny M) a pomocného pole (množiny MP).



Obrázek 12: Analýza zdola nahoru pomocí EOL systému

4.3 Zhodnocení modifikace

Tato zdánlivě malá modifikace umožňuje využít výhod EOL systémů bez změny řádu časové složitosti. Již při prvním pohledu na výsledný algoritmus a jeho případné srovnání s algoritmy klasickými je vidět, že jediné časové navýšení je režie překlápění a mazání pomocné úrovně. Stejně tak i paměťová složitost, která již více závisí na konkrétní implementaci, se zvýší pouze o tento přidaný prvek.

Jde tedy o relativně efektivní algoritmus, který se svou mocností nachází mezi bezkontextovými a kontextovými jazyky (viz. *obrázek 10*). Jako příklad jazyka, který není bezkontextový, a přesto ho lze analyzovat tímto systémem, si můžeme uvést $L(G) = \{a^{2^n}, n \geq 1\} \cup \{b^{2^n}, n \geq 1\}$. Tento jazyk můžeme generovat gramatikou z *tabulky 3* (pozn. další relevantní jazyky a gramatiky jsou přiloženy jako ukázkové vstupní data na CD). Splnili jsme tedy cíl, který jsme si zadali v úvodu kapitoly 4.

Gramatika G generující jazyk $L(G) = \{a^{2^n}, n \geq 1\} \cup \{b^{2^n}, n \geq 1\}$:

- $S \rightarrow a$
- $S \rightarrow b$
- $a \rightarrow aa$
- $b \rightarrow bb$

(S je počáteční token)

Tabulka 3: Příklad gramatiky a jazyka definovaných EOL systémem

Pokud budeme řešit problém spadající do této třídy složitosti jazyka, můžeme algoritmus bez problémů nasadit v praxi. Abychom si to dokázali, vybral jsem právě tuto část z mojí bakalářské práce k reálné implementaci. Její popis nalezneme dále.

4.4 Implementace

Stejně jako jsme v teoretické rovině vycházeli z definic bezkontextových algoritmů, ani při implementaci nebude příliš složité upravit již stávající algoritmy pro EOL systémy. Pokusíme se dodržet co největší podobnost s formálním algoritmem.

Implementační jazyk je C++. Tato volba vznikla především z důvodu skrytí režie operací nad řetězci a kontejnery, které by zastínily samotný algoritmus. Naopak použitý neobjektový přístup je zvolen proto, abychom se co nejvíce přiblížili chápaní procesu analýzy z pohledu člověka.

Program nejprve načte gramatiku a řetězec tokenů. Pokud načtení proběhlo v pořádku, spustí se samotná analýza. Výstupem programu bude pouze rozhodnutí, zda řetězec patří do jazyka generovaného gramatikou či nikoliv. Více informací je popsáno v manuálu v příloze 1 (kapitola 8.2).

Zdrojový kód je rozdělený do 4 hlavních modulů (pomineme-li modul pro výpis chybových hlášení a nápovědy): Hlavní, Gramatika, Řetězec, Analýza. Mezi těmito moduly je porušeno zapouzdření jednou globální proměnnou, která určuje úroveň reportování. Tato možnost je nastavitelná uživatelem a pracuje s ní většina funkcí, proto v této ukázkové aplikaci, u které se nepočítá s nějakým dalším vývojem do budoucna, bylo zvoleno jinak ne příliš vhodné řešení. Nám to však zajistí větší přehlednost mezi parametry funkcí, neboť každý parametr se bude týkat přímo chodu programu.

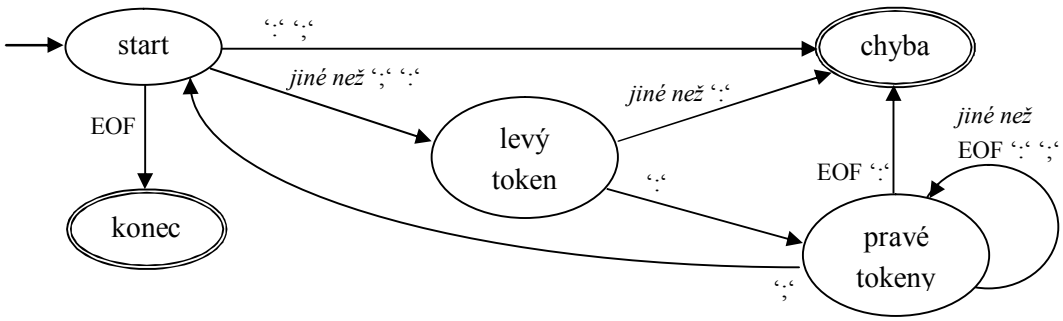
4.4.1 Modul Hlavní

Tento modul je základem programu, nikoliv však analýzy. Jeho úkolem je zkontrolovat parametry programu a postupně přejít mezi jednotlivými fázemi. Ty tvoří načtení gramatiky, načtení řetězce a analýza zdola nahoru EOL systémem. Pokud se některá z těchto fází nepovede, modul nese odpovědnost za zajištění hlášení uživateli, dealokaci paměti a správné ukončení programu.

4.4.2 Modul Gramatika

Modul Gramatika je zodpovědný za poskytnutí všech potřebných typů a funkcí k operacím nad gramatikou. Zahrnuje to načtení gramatiky, její výpis a vyhledávání pravidel dle zvolených kritérií.

Načtení probíhá po slovech ze zadaného souboru. Využívá se konečného automatu z *obrázku 13*, díky kterému máme zajištěnu kontrolu syntaxe vstupního souboru. Bílé znaky jsou ignorovány. Během načítání je automaticky generován slovník tokenů, neboť řetězcová reprezentace by byla neefektivní. Každý token má tedy odpovídající číselný index, pod kterým se s ním pracuje dále.



Obrázek 13: Konečný automat pro gramatiku

Implementaci automatu nalezneme v příloze 2 (kapitola 8.3). Můžeme si všimnout, že nikde nerozlišujeme, zda se jedná o terminály či neterminály. Je tedy použit druhý přístup z kapitoly 4.1, kdy každý token reprezentuje zároveň i symbol a může být obsažen v řetězci.

4.4.3 Modul Řetězec

Tento modul, obdobně jako modul Gramatika, poskytuje základní typy a funkce pro práci s řetězcem. Umožňuje alokaci a dealokaci prostoru pro řetězec, jeho načtení, vymazání i výpis na výstup.

Načítání řetězce probíhá ze standardního vstupu a je ukončeno znakem konec souboru (tokeny jsou od sebe odděleny bílými znaky). Řetězec je automaticky překládán na číselnou reprezentaci (podle slovníku vytvořeného gramatickým modulem), přímá textová podoba tedy uložena není (šlo by ji ale zpětným překladem ze slovníku vytvořit). Díky tomu dochází zároveň ke kontrole, zda jsme nenačetli token, který není v gramatice. Pokud ano, řetězec bude zamítnut bez zahájení samotné analýzy.

Řetězec je uložen přímo do struktury, která se bude používat v analýze. Jednotlivé části tedy neleží v paměti za sebou, ale je zde alokovaný prostor pro aplikaci pravidel a reprezentaci pozice tokenu nad řetězcem. Více o této struktuře v kapitole 4.4.5.

4.4.4 Modul Analýza

Rozhraní tohoto modulu tvoří pouze jediná funkce, která zajišťuje celý proces analýzy řetězce. Využívá k tomu gramatiky a gramatických funkcí z externího modulu a vyžaduje řetězec tokenů v připravené formě k analýze (tj. tak, jak ho při načítání upraví a uloží modul Řetězec).

Činnost modulu pro řetězec x a gramatiku G = (T, N, P, S) lze formálně zapsat tímto algoritmem:

- Vytvoříme pole M[i,j,k], pro $1 \leq i \leq j \leq n$, kde $n = |x|$, $k = |T \cap N|$ a značí jeden z tokenů; hodnoty všech prvků v tomto poli nastavíme na 0
- Pro každý znak a na pozici i nastavíme M[i,1,a] = 1
- Opakujeme, dokud jsou změny a zároveň M[1,n,S] ≠ 1
  - → Vytvořme pomocné pole MP[i,j,k] = ∅
  - → Pokud najdeme posloupnost tokenů β takovou, že β[g] je g-tý prvek posloupnosti a pro všechna g, kdy $1 \leq g \leq |\beta|-1$, platí, že M[$i,j,$β[g]] ≠ 1 a M[$j+1,k,$β[g+1]] ≠ 1 a $\alpha \rightarrow \beta \in$ P, pak MP[i,k,α] = 1
  - → M = MP (překlopíme pole MP do M)
- Slovo náleží do jazyka L(G), jestliže M[n,n,S] = 1

Vnitřní struktura modulu je však od tohoto algoritmu odlišná. První dva body za nás vytvoří již moduly Hlavní a Řetězec, my se tedy o ně nestaráme a pouze je předpokládáme. Dále analýza není tvořena jedinou funkcí, jak by se mohlo zdát z rozhranní modulu, ale je rozdělena do několika funkcí zanořených tak, aby byla čtenáři kódu co nejvíce patrná podobnost s formálním zápisem.

Proces je ukončen, pokud je nalezen počáteční token S nad celým řetězcem (je v gramatice povinný), nebo pokud se stav neměnní. Druhá část podmínky velmi záludná, neboť stav se může zdánlivě měnit, ale ve skutečnosti může proces například cyklit mezi třemi stavy dokola. Odhalit stagnaci

takového typu je docela náročné, proto jsem implementoval pouze rozpoznání prázdného stavu a omezení na maximum iterací analýzy. Nehrozí-li cyklus stavů, tak u řetězce, který do jazyka nepatří, je díky samostatným úrovním docela rychlá konvergence k prázdnému stavu.

4.4.5 Návrh struktur

Jak již bylo zmíněno, tokeny reprezentujeme jako číselné hodnoty, které přiděluje slovník. Abychom však dodrželi ortogonální návrh a umožnili změnit typ tokenu z jednoho místa, vytvoříme pro něj alias.

```
typedef unsigned int T_INDEX;
```

Pravidlo musí obsahovat právě jeden token na levé straně a posloupnost tokenů na straně pravé.

```
typedef struct S_PRAVIDLO {
      T_INDEX leva;
      vector<T_INDEX> prava;
    } T_PRAVIDLO;
```

Gramatiku definovanou uživatelem (uživatel nastavuje vše kromě počátečního tokenu, který je předvolen na ‘S‘) uložíme jako posloupnost pravidel do jednosměrného kontejneru.

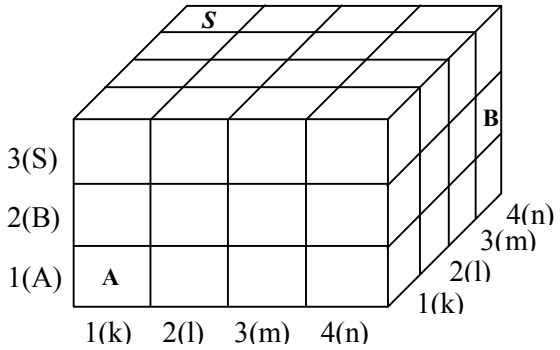
```
typedef vector<T_PRAVIDLO> T_GRAMATIKA;
```

Řetězec je příchozí posloupnost tokenů. Ačkoliv se zdá, že by bylo vhodné ho uložit do nějakého lineárního kontejneru, nebyl takový způsob zvolen. Lineární uložení nám pouze zachová řetězec ve tvaru, v jakém byl vložen, ale jediný krok v analýze, kde takový tvar potřebujeme, je při inicializaci. Proto je rovnou během načítání prováděna inicializační procedura a výstupem je přímo trojrozměrné pole, symbolizující aktuální stav procesu analýzy.

```
typedef struct S_RETEZEC {
      bool*** stav;
      unsigned int delka;
      unsigned int n3;
    } T_RETEZEC;
```

Abychom pochopili, proč právě tři rozměry, musíme si uvědomit, co symbolizují. Naším cílem je možnost reprezentace pozic tokenů ve stromě, neboli potřebujeme říct, že určitý token na určité pozici v poli reprezentuje nějaký rozsah v řetězci. První dva rozměry tvoří klasický kartézský součin nad řetězcem (mají tedy stejnou délku jako řetězec) a definují tak všechny možné podřetězce. Poslední rozměr (n3) je stejně dlouhý jako počet různých tokenů a umožní na jednu pozici umístit více než jeden token (využíváme zde jejich převedenou číselnou hodnotu).

Lépe to objasní jednoduchý příklad a *obrázek 14*. Mějme tokeny S (1), A (2), B (3) a řetězec „klmn“. Při indexování pole od 1 bude zápis M[1][3][2] = *true* znamenat, že token A reprezentuje 1. až 3. znak v řetězci (tedy „klm”). Naopak při M[3][3][3] = *true* zahrnuje token B pouze 3. znak „m”.



Obrázek 14: třírozměrné pole zobrazující token A nad prvním znakem „k“, token B nad posledním znakem „n“ a token S nad celým řetězcem „klmn“

4.4.6 Základní funkce analýzy

Popis základních funkcí, které se podílejí na procesu analýzy (jsou popsány postupně po nejvíce zanořenou). Jejich implementaci můžeme v příloze 2 (kapitola 8.3) nebo na přiloženém CD-ROM.

- `TCHYBY analyzuj(T_GRAMATIKA &g, T_RETEZEC &r, unsigned int maxi)`

  Funkce z modulu Analýza. Jde o hlavní cyklus analýzy. Provádí postupně tři kroky: aplikování gramatiky na aktuální stav (výsledky pravidel se ukládají do pomocného stavu), překlopení pomocného stavu do aktuálního a počítání iterací. Končí, pokud našli derivační strom nebo je aktuální stav prázdný nebo jsme přesáhli maximum iterací.

- `bool aplikuj_gram(T_GRAMATIKA &g,T_RETEZEC &r,T_RETEZEC &m)`

  Funkce z modulu Analýza. Právě tato funkce se stará o výběr pravidel k aplikaci na tokeny v aktuálním stavu. Postupně prochází všechny umístěné tokeny a hledá pravidla, jejichž pravá strana na ně začíná. Pokud takové nalezne, pokusí se ho aplikovat (viz. funkce zkus_aplikovat).

- `bool dej_dalsi(T_GRAM_IT &it, T_GRAM_IT end,T_INDEX start)`

  Funkce z modulu Gramatika. Vrátí první následující pravidlo, které začíná na zvolený token. Pokud takové nenajde, vrátí konec kontejneru s gramatikou.

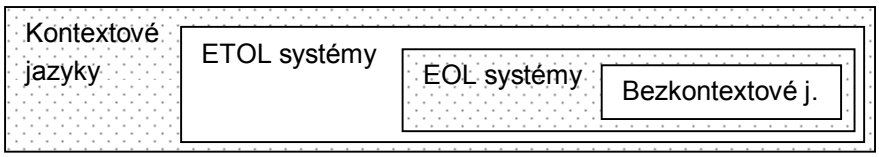
- `bool zkus_aplikovat(T_GRAM_IT &it, T_RETEZEC &r, T_RETEZEC &m, unsigned int i,unsigned int j))`

  Funkce z modulu Analýza. Její hlavní náplň je pokusit se aplikovat vybrané pravidlo na část aktuálního stavu začínající určeným tokenem. Pokud je aplikace možná, provede ji a výsledek uloží do pomocného stavu.

5 Modifikace s využitím ETOL systému

Pomocí EOL systémů jsme vytvořili mocnější analytický nástroj, než jsou bezkontextové gramatiky, ale pořád existuje velké množství jazyků, které nejsme schopni tímto způsobem analyzovat. Když budeme podrobněji studovat problematiku L-systémů, objevíme, že nad EOL systémy existují ještě určitá rozšíření. Jedním z nich jsou takzvané ETOL systémy, které pracují s více gramatickými množinami zároveň. Pokusíme se tedy navrhnout postupy analýzy založené právě na těchto systémech.

Abychom si lépe uvědomili, čeho použitím ETOL systémů dosáhneme, podívejme se na *obrázek 15*, kde máme přehledně zobrazeno srovnání mocností s nám již známými systémy (vycházíme z Chomského klasifikace jazyků z kapitoly 2.1.5) [3].



Obrázek 15: zařazení ETOL systémů

5.1 ETOL systém

Stejně jako u EOL systémů, k přesné definici ETOL systémů potřebujeme znát nejprve L-systémy. Jelikož jsme ale s EOL systémy již pracovali a prakticky tvoří předstupeň našeho problému, pokusme se definovat principy právě na srovnání s nimi. Celou definici i s doplňujícími informacemi lze opět najít v disertační práci *L-Systems: New results and application* od PhD. Ing. Milana Kolky na straně 12 [3].

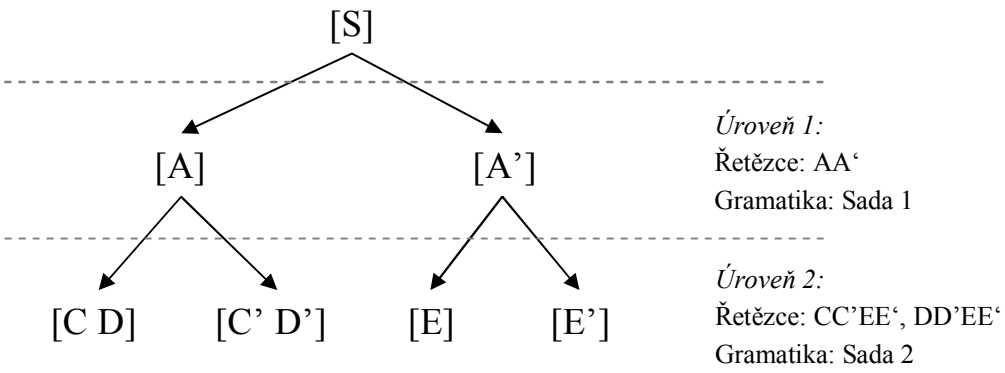
EOL systémy obsahují vždy jednu množinu s gramatickými pravidly, kterou používají pro každý derivační krok. ETOL systémy nám naopak umožňují mít těchto množin několik a při vytváření každé úrovně derivačního stromu se můžeme rozhodnout, kterou z množin použijeme. V této úrovni lze aplikovat pravidla pouze ze zvolené množiny. Pro každou úroveň můžeme zvolit jinou množinu, ze které budeme pravidla vybírat, stejně jako pro všechny úrovně lze vybrat množinu stejnou. Ostatní vlastnosti mají EOL a ETOL systémy totožné.

Pokud máme ETOL systém právě s jednou množinou pravidel, budou jeho vlastnosti i analytické schopnosti shodné s EOL systémem obsahujícím identické pravidla. Je tedy patrné, že EOL systémy jsou pouze speciálním případem těchto systémů.

Rekapitulace vlastností spolu s používáním několika množin pravidel je na *obrázku 16*. V příkladu předpokládejme tyto dvě sady pravidel:

Sada 1: S → A A‘, A → B, A‘ → B‘

Sada 2: S → B B‘, A → C C‘, A → D D‘, A‘ → E E ‘

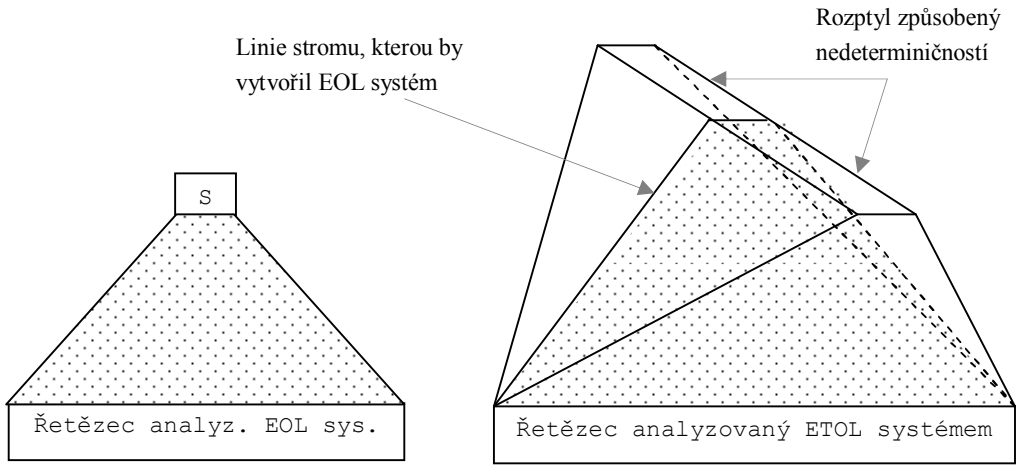


Obrázek 16: Sestavení ETOL systému

5.2 Použití v syntaktické analýze

Hlavní problém při použití více množin s gramatikou je, jak rozpoznat, kterou gramatiku máme vybrat. Pokud bychom chtěli zachovat determinismus, dostaneme se do velkých obtíží a pravděpodobně bychom snížili mocnost algoritmu (tj. byl by použitelný pouze pro určité gramatiky, obdobně jako tomu je u LL-gramatik v porovnání s bezkontextovými).

Pokusme se však zamyslet, co by bylo nutné zajistit pro nedeterministický přístup v algoritmu zdola nahoru. Stavový prostor, který musíme prohledat, je zobrazen na *obrázku 17*.

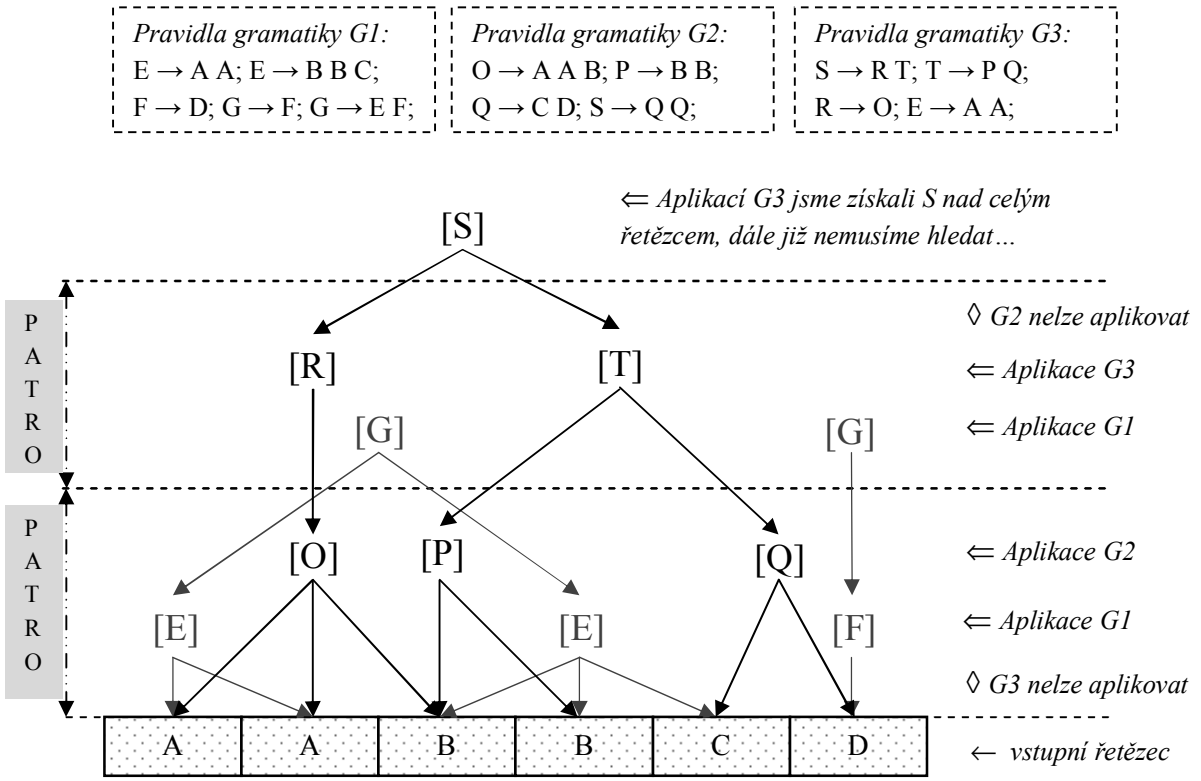


Obrázek 17: Stavový prostor EOL a ETOL analýzy

Evidentně se nyní již jedná o trojrozměrný problém a také tak s ním musíme zacházet. Kořen stromu s počátečním tokenem S může ležet kdekoliv na vrchní hraně tohoto čtyřstěnu. Hrana je vlastně tvořena samými alternativními vrcholy, a pokud je alespoň jeden shodný se S, řetězec patří do jazyka. Dva rozměry umíme vyřešit lineárně (EOL systém), proto musíme prohledávat stavový prostor do rozměru třetího – a prohledávání stavového prostoru již je klasická algoritmická úloha.

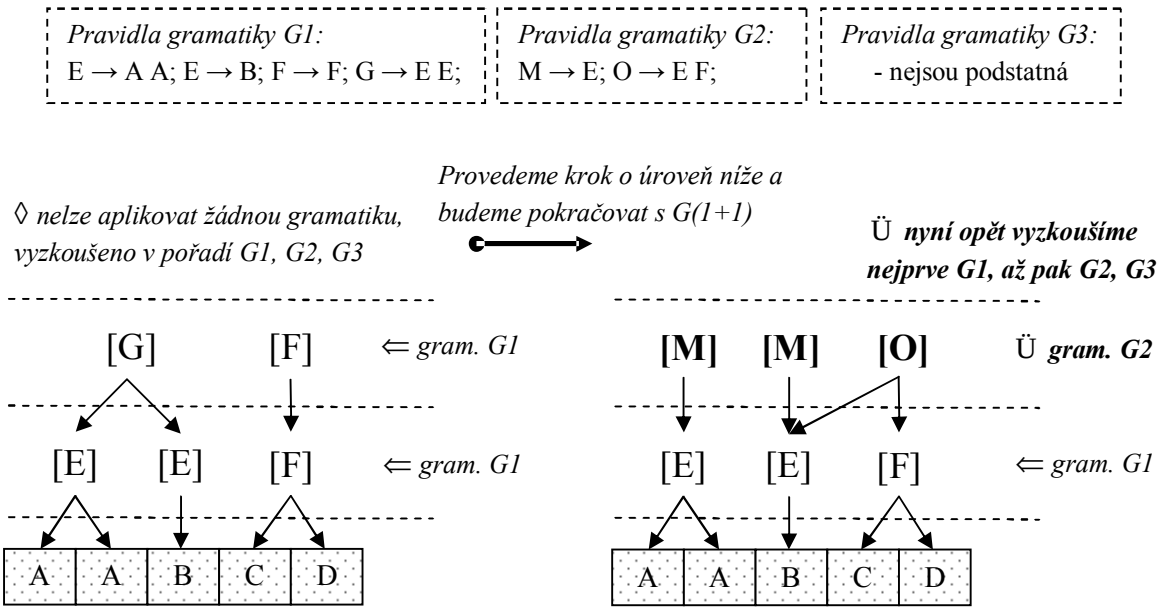
Základní metody prohledávání stavového prostoru jsou prohledávání do hloubky a do šířky. Obě metody můžeme nasadit i zde a těžit tak ze specifických vlastností a výhod zvolené varianty. Při prohledávání do šířky zabereme v paměti více místa, ale získáme tím optimálnější časové parametry a především s jistotou nalezneme nejnižší derivační strom. Prohledávání do hloubky naopak s pamětí nebude mít problém, musíme se však více soustředit na konečnost jednotlivých pokusů a nalezené řešení nemusí být nejoptimálnější (což nás na druhou stranu mnohdy nezajímá, v některých řešených problémech záleží jenom na tom, zda strom sestrojit lze). Samozřejmě existují i další použitelné metody, jako například iterační prohledávání do hloubky, avšak seznámení se s výčtem prohledávacích algoritmů je již mimo téma naší práce a ponecháme to případně na čtenáři.

Při prohledávání do šířky aplikujeme na každou úroveň ve stromě všechny sady gramatik z ETOL systému, které dokážou vytvořit celou správnou další úroveň stromu (platí zde všechny pravidla vytváření úrovní jako u EOL systémů, viz. algoritmus z kapitoly 4.2). Výsledné úrovně z každé aplikované gramatiky je nutno uložit do vlastních stavů – tím rozvětvíme strom do šířky a vznikne nám množina stavů reprezentující všechny alternativy jedné úrovně ve stromě – nazvěme si to „patro“ stromu. Pokud nás nebude zajímat tvar stromu, není nutné si v patře u každé alternativy pamatovat, z jaké větve vznikly, ale můžeme celé patro brát za jakousi množinu množin. Takto budeme dokola aplikovat všechny možné gramatiky na celá patra stromu, dokud se bude něco měnit anebo dokud alespoň v jedné alternativní úrovni libovolného patra nezískáme počáteční token S nad celým řetězcem (obdobné ukončující podmínky jako u EOL systémů). Ilustrační příklad k procesu nalezneme na *obrázku 18*.



Obrázek 18: Prohledávání do šířky

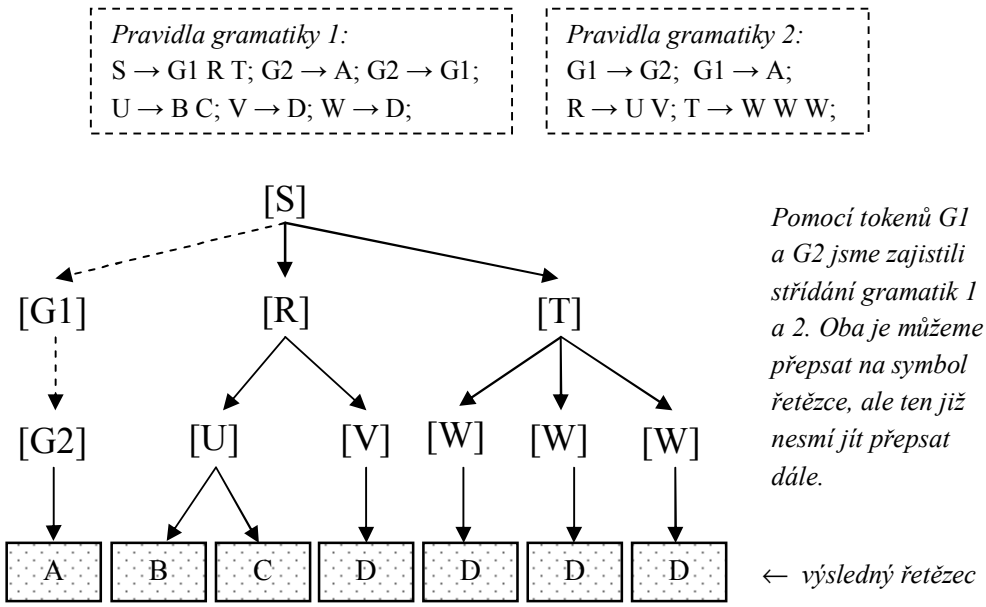
Prohledávání do hloubky je, co se týče tvorby úrovní, EOL systémům ještě podobnější. Jediný rozdíl bude ve výběru sady gramatiky a co dělat, pokud narazíme na „slepou ulici“. V takovém případě bychom v původním postupu prohlásili řetězec za neplatný (nepatřící do jazyka generovaného gramatikou), nyní však uděláme v derivačním stromu „krok zpět“ (neboli krok o úroveň níže) a vyzkoušíme k aplikaci jinou gramatiku než před tím. Proces budeme opakovat, dokud se máme kam vracet nebo existuje nějaká nevyzkoušená gramatika – nebo dokud nad celým řetězcem nezískáme počáteční token S. Abychom věděli, jakou gramatiku máme nyní vyzkoušet, doporučuji jim přidělit indexy. U každé úrovně – od aktuální níže – si poté stačí jen pamatovat index již vyzkoušené gramatiky (začíná se od první). Při návratu zpět tento index inkrementujeme, a pokud existuje příslušná gramatika (s tímto indexem), použijeme právě ji. Pokud ne, vracíme se opět o krok zpět. Postup je ilustračně naznačen na *obrázku 19*.



Obrázek 19: Prohledávání do šířky

Konvergence algoritmů se bude odvíjet od zvolené gramatiky. Čím méně budou obsahovat stejných a podobných pravidel, tím méně bude strom růst do šířky. V extrémním případě, kdy tokeny z pravých stran pravidel jsou pro každou gramatiku unikátních (opět vůči pravým stranám, kdyby byli unikátní i vůči stranám levým, neměly by gramatiky jak na sebe navazovat), dosáhli bychom deterministického postupu. K determinizmu postačuje, aby v každé gramatice bylo alespoň jedno takové unikátní pravidlo a bylo vyžadováno při každé aplikaci gramatiky. Ačkoliv tato podmínka vypadá jednoduše, dosáhnutí takového stavu bude v mnohých případech neproveditelné.

Existuje speciální případ, kdy by determinizace byla jednoduše uskutečnitelná. Pokud bychom chtěli, aby se gramatiky střídaly v předem dané posloupnosti, můžeme do každé přidat extra pravidlo. Generovalo by token, na který by bylo možné navázat pouze extra pravidlem z příslušné další gramatiky. Tento extra rozvoj by bylo nutné zahájit již přímo z počátečního tokenu S. Na průběh takového procesu se můžeme podívat na *obrázku 20*. V praxi však případy, kde by takové omezení bylo přípustné, nalezneme jen výjimečně; většinou naopak potřebujeme právě nedeterministické střídání gramatik.



Obrázek 20: Deterministické opakování gramatik

5.3 Zhodnocení modifikace

ETOL systémy jsou relativně mocný prostředek, který nám umožňuje pracovat najednou s více sadami gramatik. Naším cílem bylo navrhnout možnosti jejich využití v syntaktické analýze a ty jsme také nastínili. Hlubší průzkum těchto možností necháme již na čtenáři nebo případné navazující práci (například oblast determinizace algoritmů může skrývat ještě značné možnosti).

Při analýze řetězce zdola nahoru jsme si zmínili dva základní algoritmické přístupy, které by oba měly být použitelné i v praxi. Jejich vyšší časová náročnost je dána nutností prozkoumat velký stavový prostor a pátrat po správné variantě cesty k cíli mezi mnoha alternativami. Přesto se mohou vyskytnout problémy, kde nebude čas kritickou záležitostí nebo nebudeme pracovat s velkým množstvím gramatik (čím méně gramatik, tím méně bude alternativních cest), pak se nemusíme bát nasadit výše zmíněné postupy a využít tak větší analytické síly, kterou nám ETOL systémy přinášejí.

6 Závěr práce

V úvodu práce jsme si vytyčili několik cílů, které jsme se pokusili v průběhu práce splnit. Nakolik se nám to podařilo, jsme si podrobně zhodnotili vždy v závěru odpovídající kapitoly. Pokud mám hodnotit úspěch práce jako celku, myslím si, že pěkně shrnula základní problematiku kolem využití EOL a ETOL systémů v syntaktické analýze, nastínila možné přístupy a řešení problémů. V části týkající se modifikace CYK algoritmu jsme nejen odhalili potíže vznikající při snaze o jeho otočení, ale navrhli jsme několik variant jejich řešení. Čtenář tak získal širší nadhled nad problematikou, který by mu měl přinést inspiraci k řešení i jiných úkolů.

Při psaní práce jsem si rozšířil znalosti bezkontextových gramatik a některých variant L-systémů nad rozsah učiva bakalářského stupně. Propojil jsem v ní znalosti z teorie formálních jazyků se znalostmi prohledávacích algoritmů a stromových struktur. Nakonec jsem implementoval i ukázkovou aplikaci, která je vytvořena nejen pro testování algoritmu, ale díky snaze o přehlednost a velkou podobnost s formálním algoritmem také k samotnému pochopení činnosti algoritmu.

V práci se také naskýtá několik možností k dalšímu výzkumu a pokračování. Pomineme-li způsoby syntaktické analýzy, tak jsme v kapitole 4.4.4 zmínili problém s rozpoznáním neměnného stavu procesu s ohledem na možné zacyklení. Odhalit takovýto stav může být velmi obtížné, neboť neznáme délku cyklu a ta se může blížit i celé výšce derivačního stromu.

Ani oblast ETOL systémů a jejich využití v syntaktické analýze není zatím důkladně prozkoumána. Například možnosti částečné nebo úplné determinizace procesu jsme ponechali spíše na budoucí studium a výzkum. Podařilo se nám navrhnout základní přístupy k použití těchto systémů, které jsou úplné a využitelné v praxi. Avšak také tato oblast nedeterministického přístupu, kdy prohledáváme široký stavový prostor, by si zasloužila hlubší prozkoumání a bylo by vhodné na ni navázat například v diplomové práci.

7 Literatura

[1] MEDUNA, Alexander, LUKÁŠ, Roman. *Formální jazyky a překladače* [přednášky online]. FIT VUT, Brno, aktualizováno 2006-09-19 [cit. 2009-03-15]. Dostupné na URL: <https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/>.

[2] MEDUNA, Alexander. *Elements of Compiler Design*. Boca Raton : Auerbach Publications, 2008. xiii, 286 s. ISBN 978-1-4200-6323-3.

[3] KOLKA, Milan. *L-Systémy: Nové výsledky a aplikace*. [s.l.], 2003. 110 s. Fakulta informačních technologií, Vysoké učení technické v Brně. Vedoucí dizertační práce Doc. RNDr. Alexander Meduna, CSc.

[4] BLATNÝ, Petr. *Syntaktická analýza založená na gramatikách s rozptýleným kontextem*. [s.l.], 2004. 46 s., 1 CD-ROM. Fakulta informačních technologií, Vysoké učení technické v Brně. Vedoucí diplomové práce Doc. RNDr. Alexander Meduna, CSc.

8 Přílohy

Tato práce obsahuje jak textové přílohy, tak externí CD-ROM s elektronickou verzí práce a doplňujícími soubory. Textové přílohy se postupně nacházejí dále v této kapitole.

8.1 Seznam příloh

- Příloha 1.: Manuál k EOL analyzátoru
- Příloha 2.: Vybrané části zdrojového kódu EOL analyzátoru
- Příloha 3.: CD-ROM s elektronickou verzí práce, zdrojovými texty a praktickými příklady

8.2 Příloha 1: Manuál k EOL analyzátoru

Program slouží k syntaktické analýze řetězce pomocí EOL systému metodou zdola nahoru. Očekává od uživatele gramatiku jazyka a vstupní řetězec, pro který rozhodne o jeho příslušnosti do jazyka. Je plně funkční na operačních systémech MS Windows i Linux.

Synopsis programu:

```
program [-v|-d|-ed] [-max <kolik>] <gramatika>
```

První nepovinný parametr specifikuje úroveň informačních výpisů. Ty jsou stejně jako veškerá chybová hlášení a nápověda psány na standardní chybový výstup. Jednotlivé varianty značí režimy: Verbous, Debug, Experimental Debug – v tomto pořadí roste i množství vypsaných informací. Experimental Debug je režim, který se snaží vykreslovat do tabulky stav po jednotlivých iteracích algoritmu. Proto není doporučeno ho používat s dlouhými řetězci a velkým počtem různých tokenů v gramatice.

Druhý nepovinný parametr umožňuje uživateli omezit maximální počet iterací algoritmu. Pokud nastaven explicitně, program se zastaví po 500 iteracích. Zabraňuje to nekonečným cyklům v analýze.

Třetí parametr je povinný a je jím název (popřípadě i cesta) k souboru s gramatikou. Možnosti zápisu gramatiky jsou popsány níže.

Řetězec je očekáván na standardním vstupu jako posloupnost symbolů oddělena libovolným množstvím bílých znaků (mezery, tabulátory, konce řádku…) a ukončena znakem konec souboru (ručně: Linux Ctrl+D, Windows Ctrl+Z).

Rozhodnutí o příslušnosti řetězce do jazyka generovaného gramatikou je vypsáno jako jediná informace na standardní výstup (umožňuje to jednodušší řetězení programů či zachytávání výstupu). V případě příslušnosti do jazyka je ve tvaru „*accept*“, jinak „*reject*“ (konstanta ve zdrojovém souboru, lze jednoduše redefinovat).

Příklady spuštění:

- `program gramatika.txt < retezec.txt`
  – nejkratší varianta zápisu, žádné informační výpisy

- `program -v gramatika.txt < retezec.txt 2>report.txt`
  – střední úroveň reportu s jeho přesměrováním do souboru

- `program -d -max 100 gramatika.txt < retezec.txt >vysledek.txt`
  – vysoká úroveň reportu, omezení na 100 iterací a výsledek analýzy přesměrován do souboru

Soubor s gramatikou:

Soubor obsahuje zapsanou posloupnost pravidel v textové formě. K zajištění co největší přehlednosti umožňuje i psaní řádkových komentářů. Podmínky správného zapsání pravidla jsou:

- Všechny tokeny jsou jednoslovné (nesmí obsahovat bílé znaky) a existují 4 rezervované: „ // “, „**S**“, „ **:** “, „ **;** “ (dvojtečka a středník). Symboly z těchto tokenů se mohou objevit v uživatelských tokenech, ale musí být v kombinaci s jinými znaky.
- Tokeny jsou odděleny bílými znaky (jejich počet nehraje roli, povinný alespoň jeden).
- Token „**S**“ je v gramatice povinný a značí počáteční token.
- Pravidlo se stává z levé a pravé strany oddělené tokenem „ **:** “.
- Levá strana je tvořena právě jedním tokenem. V gramatice musí existovat pravidlo, které má token „**S**“ na této straně.
- Pravá strana obsahuje posloupnost jednoho nebo více tokenů ukončenou tokenem „ **;** “.
- Celá sada pravidel je ukončena znakem konec souboru.
- Token „ // “ uvozuje řádkový komentář. Při jeho nalezení bude on i zbytek řádku zahozen.

Pokud nám z nějakého důvodu nevyhovuje pojmenování tokenu symbolizujícího počáteční token znakem „S“ (např. kvůli specifickému pojmenování tokenů v řešeném problému), můžeme si ho redefinovat přidáním pravidla ve tvaru „*S : novy_pocatecni_token ;*“.

Pro lepší pochopení zápisu si ukážeme, jak by vypadal soubor s gramatikou z *tabulky 3* (kapitola 4.3).

```
// analyzuje jazyk (a^2^n|n>=0)U(b^2^n|n>=0)
S : a ;
S : b ;
a : a a ;
b : b b ;
```

Další ukázky souborů s gramatikami, stejně jako elektronickou verzi tohoto manuálu, naleznete na přiloženém CD-ROM.

8.3 Příloha 2: Vybrané části zdrojového kódu EOL analyzátoru

Ze souboru gramatika.cpp

Implementace konečného automatu k načtení gramatiky z kapitoly 4.4.2. V průběhu načítání se automaticky vytváří i slovník tokenů a symboly se jím překládají na číselnou hodnotu.

```
TCHYBY nacti_gramatiku(T_GRAMATIKA &g, T_SLOVNIK_TOKENU &s, char *f)
{ // otevreni a test souboru
  ifstream f_gram(f, ifstream::in);
  <osetreni chyby>

  // nacteme pravidla
  string token;                          // retezova hodnota tokenu
  T_INDEX it;                            // ciselne vyjadreni tokenu
  enum T_stav {Start, Levy, Pravy, Konec, Chyba}; // stavy automatu
  T_stav stav=Start;                              // aktualni stav
  T_INDEX start=pridej_token(s,string(START)); // ulozime si start. symbol

  bool levy_start=false;   // priznak, zda se na leve strane vyskytl start
  T_PRAVIDLO pravidlo;     // sestavovane pravidlo gramatiky

  // nacitani konecnym automatem
  while (f_gram.good() && stav!=Konec && stav!=Chyba)
  { f_gram >> token;     // nacteme tokeny ze souboru
    if (token==Komentar) // narazili jsme na komentar - vypustime radek
    { f_gram.ignore(4000,'\n');  continue;  }

    switch (stav)
    { case Start:   // stav - ocekavame pravidlo (předchozí dokonceno)
            if (token==LevyXPravy) stav=Chyba;       // neocekavany symbol
            else if (token==TokenKonec) stav=Chyba;  // neocekavany symbol
            if (f_gram.eof()) stav=Konec;
            else
            { it=pridej_token(s,token);  // pridame token do slovniku
              if (!levy_start && it == start) levy_start=true;
              pravidlo.leva=it;          // vlozime levy token do pravidla
              pravidlo.prava.clear();    // vycistime pravo cast  pravidla
              stav = Levy;               // - vlevo hotovo
            }
            break;

      case Levy:    // stav - mame levou stranu pravidla a cekame oddelovac
            if (token.compare(LevyXPravy)!=0) stav=Chyba;
            else stav = Pravy;      // oddelovac - ocekavame pravou stranu
            break;
```

```
      case Pravy:   // stav - sestavujeme pravou stranu pravidla
             if (token.compare(LevyXPravy)==0) stav=Chyba;
             else if (token.compare(TokenKonec)==0)  // pravidlo ukonceno
             { if (pravidlo.prava.size()<1) stav=Chyba;
               else                         // jestli je pavidlo ok,
               { g.push_back(pravidlo);     // pridame ho do gramatiky
                 stav=Start;
               }
             }
             else   // prisel token do prave strany pravidla
             { it=pridej_token(s,token);     // prelozime token ve slovniku
               pravidlo.prava.push_back(it); // pridame ho na pravou stranu
             }
             break;

      default: break;
    } // switch
  } // while

  f_gram.close(); // zavreme soubor s gramatikou - vse nacteno

  // kontrola vysledku nactinani automatem
  if (stav!=Konec && stav!=Start) return E_GRAMATIKA;

  // kontrola, zda je startovni symbol na leve strane nejakeho pravidla
  if (!levy_start) return E_G_START;

  return E_OK;
} // nacti_gramatiku
```

Funkce najde další pravidlo v gramatice za iterátorem „it“, které začíná na symbol „start“.

```
bool dej_dalsi(T_GRAM_IT &it, T_GRAM_IT end,T_INDEX start)
{ for (it=it+1;it!=end;it++)
    if (it->prava[0]==start) return true;
  return false;
} // dej_dalsi
```

Soubor analyza.cpp

Funkce „překlopí“ mezistav iterace v analýze do aktuálního stavu a mezistav vynuluje.

```
void preklop(T_RETEZEC &r,T_RETEZEC &m)
{ for (unsigned int i=0; i<r.delka; i++)
    for (unsigned int j=i; j<r.delka; j++) // stavy, kdy j<i, nemaji smysl
      for (unsigned int k=0; k<r.n3; k++)
       { r.stav[i][j][k]=m.stav[i][j][k];
         m.stav[i][j][k]=false;
       }
} // predklop
```

Otestujeme pravidlo z parametru programu na aktuální stav, a pokud je vhodné, aplikujeme ho (výsledek uložíme do mezistavu).

```
bool zkus_aplikovat(T_GRAM_IT &it,T_RETEZEC &r,T_RETEZEC &m,
                    unsigned int i,unsigned int j)
{ unsigned int ix=i,jx=j;

  // postupne projdeme vsechny zbyvajici casti na prave strane pravidla
  for (unsigned int n=1;n<it->prava.size();n++)
  { jx=ix=jx+1; // nastavime odkud prohledavat stavovy prostor retezce
    // hledame token na moznych umistenich ve stavovem prostoru retezce
    while(jx<r.delka && !(r.stav[ix][jx][it->prava[n]-1]))
    { jx++; }
    // pokud nenasel vhodny token, tak pravidlo nesedi
    if (!(jx<r.delka)) return false;
  }
  // aplikace: rozsah pravidla zastresime hodnotou tokenu na leve strane
  return m.stav[i][jx][it->leva-1]=true;
}// zkus_aplikovat
```

Pro každý symbol řetězce najdeme všechny pravidla, jejichž pravá strana na něj začíná, a necháme je otestovat + případně aplikovat.

```
bool aplikuj_gram(T_GRAMATIKA &g,T_RETEZEC &r,T_RETEZEC &m)
{ T_GRAM_IT pravidlo;
  bool neprazdne=false;   // test, zda aplikujeme aplespon jedno pravidlo
  unsigned int n;

  for (unsigned int i=0; i<r.delka; i++)     // prochazeni retezce
    for (unsigned int j=i; j<r.delka; j++)   // stavy kdy j<i nemaji smysl
      for (unsigned int k=0; k<r.n3; k++)
        if (r.stav[i][j][k])
         {  pravidlo=g.begin();
            n=1;
            if (k+1==pravidlo->prava[0])     // osetreni prvniho pravidla
            { if (zkus_aplikovat(pravidlo,r,m,i,j)) neprazdne=true;
              n++;
            }
            while(dej_dalsi(pravidlo,g.end(),k+1)) // dalsich pravidla
            { if (zkus_aplikovat(pravidlo,r,m,i,j)) neprazdne=true;
              n++;
            }
         }

  if (neprazdne) return true;
  return false;         // zadny token nenalezen => dale nepokracovat
} // aplikuj_gram
```

Hlavní cyklus analýzy, ve kterém se provádí jednotlivé iterace. Dokud není počáteční token nad celým řetězcem, dokud jsme nedosáhli maxima iterací nebo dokud nejsme v prázdném stavu, tak aplikuje pravidla na aktuální stav a vytváří tím stav nový.

```
TCHYBY analyzuj(T_GRAMATIKA &g,T_RETEZEC &r,unsigned int maxi)
{ unsigned int n_iteraci=0;  // pocitadlo iteraci
  bool kon=false;            // nedostali jsme se do prazdneho stavu?

  // pripravime pomocny mezistav pro analyzy retezce
  // -> zajisti nam stejny level v EOL systemu
  T_RETEZEC m;
  m.delka = r.delka;
  m.n3=r.n3;

  TCHYBY err=vytvor_stav_prostor(m);
  if (err!=E_OK) return err;
  nuluj_stav_prostor(m);

  // cyklus analyzy - konec = pocatecni token pres cely řetězec a limit
  while(!r.stav[0][r.delka-1][0] && n_iteraci<maxi && !kon)
  {
    kon=!aplikuj_gram(g,r,m);   // najde pravidla pro r a aplikuje do m
    preklop(r,m);               // preklopi mezistav do retezce
    n_iteraci++;
  }

  // mezistav jiz nebude potreba - dealokace
  zrus_stavovy_prostor(m);

  // analyza vysledku
  if (r.stav[0][r.delka-1][0]) return V_ACCEPT; // retezec ACCEPT
  return V_REJECT;                              // retezec REJECT
} // analyzuj
```