VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉMY SYNTAKTICKÝCH ANALYZÁTORŮ

PARSER SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MATEJ KUNDA**
AUTHOR

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

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Kunda Matej**
Program: Informační technologie
Název: **Systémy syntaktických analyzátorů**
Parser Systems
Kategorie: Překladače

Zadání:

1. Dle instrukcí vedoucího se seznamte s vybranými pokročilými metodami syntaktické analýzy.
2. Dle instrukcí vedoucího zaveďte nové metody syntaktické analýzy, které jsou založeny na kombinaci několika dílčích metod syntaktické analýzy.
3. Studujte vlastnosti metod z bodu 2. Porovnejte jejich sílu s klasickými metodami syntaktické analýzy.
4. Dle pokynů vedoucího uvažujte alespoň 5 syntaktických struktur, včetně struktur, které nejsou bezkontextové. Popište jejich analýzu prostřednictvím modifikovaných metod z bodu 2.
5. Dle instrukcí vedoucího aplikujte metody z bodu 2 v kompilátorech. Aplikaci zaměřte na analýzu syntaktických struktur, které nejsou bezkontextové.
6. Implementujte aplikaci navrženou v bodě 5 a testujte ji v rámci stávajících nástrojů.
7. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

- Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1
- Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition), Pearson Education, 2006, ISBN 0-321-48681-1
- Meduna, A.: Automata and Languages, Springer, London, 2000, ISBN 978-1-85233-074-3
- Meduna, A.: Elements of Compiler Design, New York, Taylor & Francis, 2008

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

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

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

Abstrakt

Cieľom tejto práce je vytvorenie formálneho modelu nového gramatického systému, ktorý dokáže kombinovať niekoľko metód syntaktickej analýzy. Inšpiráciou k vytvoreniu gramatického systému boli kooperačne distribuované gramatické systémy, z ktorých sa vytvorila myšlienka komunikačných symbolov. Pomocou komunikačných symbolov dokážu komponenty gramatického systému komunikovať, a tak spolupracovať na tvorbe jednej vety. V práci som navrhol formálny model tohoto systému, aplikoval sa v syntaktickej analýze a následne implementoval. Výsledkom práce je aplikácia, ktorá funguje na princípe formalizovaného gramatického systému a kombinuje dve metódy syntaktickej analýzy.

Abstract

The goal of this thesis is to make a formal model of a new grammar system, which can combine different methods of syntax analysis. The inspiration for the making of the new grammar system were cooperating distributed grammar systems, from which an idea of communication symbols came from. With the help of communication symbols, components of the grammar system can cooperate on creating one sentential form. The thesis describes the formalization of the grammar system, its application in syntax analysis and finally its implementation. The result of this thesis is an application, which works on the principle of the formalized grammar system and combines two methods of syntax analysis.

Kľúčové slová

gramatický systém, syntaktická analýza, deterministická analýza zhora nadol, deterministická analýza zdola nahor

Keywords

grammar system, syntax analysis, deterministic top-down parsing, deterministic bottom-up parsing

Citácia

KUNDA, Matej. *Systémy syntaktických analyzátorů*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexander Meduna, CSc.

Systémy syntaktických analyzátorů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána prof. RNDr. Alexandra Medunu CSc. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Matej Kunda
11. mája 2022

Poďakovanie

Chcel by som poďakovať pánovi prof. RNDr. Alexandrovi Medunovi CSc. za vytvorenie zadania, vedenie práce a následnú pomoc pri jej tvorbe. Taktiež sa chcem poďakovať Aničke a Tiborovi za gramatickú a typografickú korektúru práce.

Obsah

Kapitola 1

Úvod

Jazyk už odpradávna tvorí neoddeliteľnú súčasť ľudského života. Človek mal postupom času potrebu komunikovať aj s počítačom, preto vytvoril jazyky programovacie. Programovacím jazykom je možné počítaču povedať, čo a ako má robiť. Tieto jazyky sú navrhnuté so striktnou a konkrétnou vetnou skladbou, ktorú nie vždy dokáže počítač pochopiť. V praxi sa jedná o jazyky nekonečné, avšak takéto jazyky je nutné popísať konečnou formou. Tento problém viedlo informatikov a matematikov k vytvoreniu gramatiky.

Gramatika je formálny model, ktorý dokáže pomocou pravidiel popisovať konečné aj nekonečné jazyky konečným spôsobom. Spočiatku tieto pravidlá boli definované veľmi všeobecne, preto v roku 1959 predstavil americký matematik Noam Avram Chomsky štyri typy gramatík rozdelené podľa ich generatívnej sily – regulárne, bezkontextové, kontextové a všeobecné. Tieto gramatiky tvoria základ v teórii formálnych jazykov v informatike a ich postupným rozširovaním došlo k vytvoreniu gramatického systému. Jedným z typov sekvenčných gramatických systémov je aj kooperačne distribuovaný (CD) gramatický systém. Takýto gramatický systém obsahuje viacero gramatík, ktoré spolu komunikujú a kooperovane vytvárajú jednu vetnú formu. Na komunikáciu medzi jednotlivými gramatikami slúžia tzv. komunikačné symboly, ktoré boli inšpiráciou k vytvoreniu nového gramatického systému v tejto práci. Podobne ako CD gramatický systém, aj systém vytvorený v tejto práci, pracuje sekvenčne a používa svoje komponenty na generovanie jednej vetnej formy. Hlavným uplatnením gramatického systému predstaveného v tejto práci je v procese syntaktickej analýzy.

Syntaktická analýza je proces overovania vetnej skladby programovacích jazykov. V procese prekladu programovacieho jazyka na strojový jazyk zastáva úlohu kontroly syntaxe jazyka. Na vykonávanie syntaktickej analýzy slúži syntaktický analyzátor, ktorý pracuje jednou z dvoch metód. Prvou metódou je syntaktická analýza zhora nadol, ktorá rozvíja vetu ako celok na menšie podčasti a ďalej na samotné slová. Druhou metódou je analýza zdola nahor, ktorá ako už názov napovedá, funguje naopak. Táto metóda spája jednotlivé slová do častí a tie spolu do výslednej vety. Gramatický systém navrhnutý v tejto práci umožňuje kombináciu viacerých metód syntaktickej analýzy, a tým zvyšuje ich silu. V priebehu analýzy dokáže meniť spomínané metódy, a to pomocou predávania kontextu medzi svojimi komponentmi.

Z pohľadu prekladača sa syntaktická analýza nachádza za lexikálnym analyzátorom. Lexikálny analyzátor prevádza vstup na postupnosť tokenov, s ktorými ďalej pracuje časť syntaktickej analýzy. Za syntaktickou analýzou sa nachádza sémantická analýza, generovanie vnútornej formy programu, optimalizácia a generovanie cieľového kódu. Tieto časti

prekladača sa starajú o overovanie sémantickej stránky programu, optimalizáciu a generovanie výsledného kódu.

Cieľom práce je vytvorenie formálneho modelu gramatického systému, ktorý bude následne použitý prakticky ako syntaktický analyzátor kombinujúci metódy zhora nadol a zdola nahor. K umožneniu praktickej implementácie takéhoto systému bolo potrebné implementovať provizorný lexikálny analyzátor vo veľmi obmedzenej forme, ktorý dokáže prevádzať vstupné lexémy na tokeny, s ktorými ďalej pracuje gramatický systém. Navrhnutý gramatický systém pre syntaktickú analýzu pracuje sekvenčne s dvomi komponentmi, z ktorých každý vykonáva odlišnú metódu.

Jadro práce tvoria štyri kapitoly. V Kapitole 2 sa nachádza vysvetlenie základných pojmov ako sú reťazec, jazyk a gramatika. S týmito pojmami ďalej pracuje Kapitola 3, ktorá formálne definuje gramatický systém a obsahuje aj demonštráciu fungovania gramatického systému na príkladoch. Nasleduje Kapitola 4, v ktorej sú opísané metódy zhora nadol a zdola nahor. Na konci tejto kapitoly sa nachádzajú praktické príklady navrhnutého gramatického systému na syntaktických štruktúrach z vytvoreného jazyka, ktorý je popísaný formou pravidiel v Prílohe A. Implementácia programu je popísaná v Kapitole 5. Nakoniec, v Kapitole 6, je spísaný záver práce.

Kapitola 2

Reťazce, jazyky a gramatiky

Cieľom tejto kapitoly je vysvetlenie základných pojmov ako sú formálny jazyk, reťazec, operácie nad jazykmi a reťazcami. Ďalej definuje jednotlivé časti gramatiky, gramatiku, derivačné kroky, ich uzávery a jazyk prijímaný gramatikou. Na konci definícií je uvedený ich zdroj. Sekcie boli inšpirované zdrojom [7].

2.1 Reťazce a jazyky

Pred uvedením samotnej definície jazyka, je najprv potrebné sa oboznámiť s pojmami ako sú *abeceda* a *reťazec*.

Definícia 2.1.1. Abeceda je konečná neprázdna množina, ktorej prvky sa nazývajú *symboly abecedy* [4].

Každá neprázdna podmnožina abecedy sa nazýva podabecedou. Ako príklad abecedy je možné uviesť latinskú abecedu obsahujúcu 52 symbolov, ktoré reprezentujú veľké a malé písmená. Symboly abecied sa môžu zlučovať a tým tvoriť reťazce.

Definícia 2.1.2. *Reťazcom* (taktiež slovom, alebo vetou) nad danou abecedou sa rozumie každá konečná postupnosť symbolov abecedy. Prázdna postupnosť symbolov, t.j. postupnosť, ktorá neobsahuje žiadny symbol, sa nazýva *prázdny reťazec*. Prázdny reťazec sa označuje písmenom ε. Formálne sa dá definovať reťazec nad abecedou Σ nasledovne:

1. Prázdny reťazec ε je reťazec nad abecedou Σ.
2. Ak je x reťazcom nad Σ a $a \in \Sigma$, potom xa je reťazec nad Σ.
3. y je reťazec nad Σ, práve vtedy a len vtedy, ak je y možné získať aplikáciou pravidiel 1 a 2 [7].

Konvencia 2.1.1. Pri práci s abecedami, symbolmi a reťazcami budeme používať nasledujúce značenie:

- veľké grécke písmená pre abecedy,
- malé latinské písmená zo začiatku abecedy (a,b,c,...) pre symboly,
- malé latinské písmená z konca abecedy (u,v,w,...) pre reťazce.

Definícia 2.1.3. Nech x a y sú reťazcami nad abecedou Σ. *Konkatenáciou* (zreťazením) reťazca x s reťazcom y vznikne reťazec xy pripojením reťazca y za reťazec x [7].

Operácia konkatenácie je asociatívna, t.j.

$$x(yz) = (xy)z,$$

nie však komutatívna,

$$xy \neq yx.$$

Každý reťazec je možné rozdeliť na časti, ktoré sa nazývajú podreťazcom, predponou a príponou.

Definícia 2.1.4. Nech w je reťazcom nad abecedou Σ. Reťazec z sa podľa [7] nazýva *podreťazcom* reťazca w, pokiaľ existujú reťazce x a y také, že $w = xzy$. Reťazec x_1 sa nazýva *prefixom* (predponou) reťazca w, ak existuje reťazec y_1 taký, že $w = x_1y_1$. Analogicky, reťazec y_2 sa nazýva *sufixom* (príponou) reťazca w, pokiaľ existuje reťazec x_2 taký, že $w = x_2y_2$. Ak platí $y_1 \neq \varepsilon$, resp. $x_2 \neq \varepsilon$, potom x_1 je *vlastný prefix*, resp. y_2 je *vlastný sufix* reťazca w.

Je zrejmé, že prefix a sufix nejakého reťazca je zároveň jeho podreťazcom. Ďalej, prázdny reťazec je podreťazcom, prefixom aj sufixom každého reťazca.

Konvencia 2.1.2. Reťazec alebo podreťazec, ktorý sa skladá práve z k výskytov symbolu a, budeme symbolicky značiť a^k. Napr.

$$a^3 = aaa,\ b^2 = bb,\ a^0 = \varepsilon.$$

Po uvedení základných definícií nasleduje definícia jazyka vyplývajúceho z [7].

Definícia 2.1.5. Nech Σ je abeceda. Symbolom Σ^* sa označuje množina všetkých reťazcov nad abecedou Σ vrátane ε, symbolom Σ^+ množina všetkých reťazcov nad Σ vynímajúc ε, t.j. $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Množina L, pre ktorú platí $L \subseteq \Sigma^*$ (prípadne $L \subseteq \Sigma^+$, ak $\varepsilon \notin L$) sa nazýva *jazykom* L nad abecedou Σ. Jazykom teda môže byť ľubovoľná podmnožina reťazcov nad danou abecedou. Reťazec $x, x \in L$ sa nazýva *vetou* (taktiež niekedy slovom) jazyka L.

Pre účely tejto práce nie je potrebné definovať operácie nad jazykmi, a iné, zložitejšie, definície. Práca sa zameriava skôr na formálne štruktúry, ktoré dokážu jazyky reprezentovať a to aj jazyky so zložitejšou štruktúrou.

2.2 Gramatiky

Jazyk je možné definovať viacerými spôsobmi. Konečné jazyky je možné definovať vymenovaním jednotlivých hodnôt, nekonečné jazyky sa dajú zapísať pomocou charakteristickej vlastnosti. Ak však ide o zložité jazyky s komplexným správaním, je prehľadnejšie a jednoduchšie použiť formálny popis pomocou gramatík. Gramatiky sú najznámejším prostriedkom pre reprezentáciu jazykov, dokážu popisovať ako jazyky konečné, tak aj nekonečné a to konečnou reprezentáciou. Využívajú dve konečné disjunktné abecedy: množinu *neterminálnych symbolov* N a množinu *terminálnych symbolov* Σ. Neterminálne symboly, skrátene neterminály, plnia úlohu pomocných premenných označujúcich určité syntaktické celky. Množina terminálnych symbolov, skrátene terminálov, je zhodná s abecedou, nad ktorou je definovaný jazyk. Množiny terminálnych a neterminálnych symbolov musia byť disjunktné.

Konvencia 2.2.1. Pre tieto množiny a ich kombinácie zavedieme v tejto práci konvenciu:

- malé písmená latinskej abecedy a, b, c, d budú označovať terminálne symboly,
- veľké písmená latinskej abecedy a ich kombinácie $A, B, \ldots, Z$ budú označovať neterminálne symboly,
- malé písmená gréckej abecedy $\alpha, \beta, \ldots, \omega$ budú označovať reťazce terminálnych a neterminálnych symbolov,
- malé písmená latinskej abecedy $u, v, \ldots, z$ budú označovať reťazce terminálnych symbolov.

Generatívna sila gramatiky vychádza z jej pravidiel. *Prepisovacie pravidlá* umožňujú ich aplikáciou generovať terminálne a neterminálne symboly a ich reťazce počínajúc od neterminálneho symbolu nazývaným počiatočný symbol gramatiky. Tieto reťazce sa nazývajú *vetnými formami*. Vetné formy tvorené iba terminálnymi symbolmi sa nazývajú *vetami*, ktoré reprezentujú vety jazyka generované gramatikou. Množina prepisovacích pravidiel je v tvare usporiadanej dvojice (α, β). Takto zapísané pravidlo stanovuje možnú substitúciu reťazca β namiesto reťazca α, ktorý sa vyskytuje ako podreťazec generovaného reťazca. Reťazec α obsahuje aspoň jeden neterminálny symbol, reťazec β je prvok z množiny $(N \cup \Sigma)^*$. Formálne vyjadrené, množina P prepisovacích pravidiel je podmnožinou kartézskeho súčinu:

$$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*. \tag{2.1}$$

Chomsky vo svojej práci [2] tento zápis prepisovacích pravidiel definuje ako gramatiku typu 0 – všeobecnú gramatiku. Gramatika typu 0 poskytuje veľkú vyjadrovaciu silu na úrovni Turingových strojov, ktorú v tejto práci nebudeme potrebovať. Práca sa obmedzuje preto na striktnejší typ gramatík typu 2 – bezkontextové gramatiky. Bezkontextové gramatiky obsahujú pravidlá v tvare usporiadanej dvojice (A, β), kde pravá strana pravidla β umožňuje substituovať neterminál A, ktorý je súčasťou generovaného reťazca. Reťazec β je rovnako ako pri gramatikách typu 0 prvkom z množiny $(N \cup \Sigma)^*$. Množinu P prepisovacích pravidiel môžeme vyjadriť formálne ako:

$$P \subseteq N \times (N \cup \Sigma)^*. \tag{2.2}$$

Ďalším typom gramatiky je gramatika typu 1, nazývaná aj kontextová gramatika. Množina pravidiel kontextovej gramatiky je formálne zapísaná ako:

$$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^* (N \cup \Sigma)^+ (N \cup \Sigma)^*. \tag{2.3}$$

Množina P ďalej môže obsahovať aj pravidlo $S \to \varepsilon$, pokiaľ sa S nikdy neobjavuje na pravej strane žiadneho pravidla.

Konvencia 2.2.2. Bez ujmy na všeobecnosti považujme vo zvyšku práce pojem *gramatika* ako vyššie definovanú gramatiku typu 2, pokiaľ nie je uvedené inak.

Konvencia 2.2.3. Pre každú usporiadanú dvojicu (A, α) zavedieme v tejto práci konvenciu zápisu prepisovacích pravidiel na $A \to \alpha$. Potom hovoríme A sa prepíše na α. Neterminál A nazývame ľavou stranou prepisovacieho pravidla, reťazec α nazývame pravou stranou prepisovacieho pravidla.

S využitím všetkých pojmov nasleduje úplná, formálna definícia gramatiky:

Definícia 2.2.1. *Gramatika* G je štvorica $G = (N, \Sigma, P, S)$, kde

- N je abeceda neterminálnych symbolov,
- Σ je abeceda terminálnych symbolov, $N \cap \Sigma = \emptyset$,
- P je konečná podmnožina kartézskeho súčinu $N \times (N \cup \Sigma)^*$ nazývaná prepisovacie pravidlá,
- $S \in N$ je počiatočný symbol gramatiky [7].

Generovanie výslednej vety prebieha postupne pomocou uplatnenia prepisovacích pravidiel. Zmena reťazca použitím prepisovacieho pravidla sa nazýva derivačný krok.

Definícia 2.2.2. Nech $G = (N, \Sigma, P, S)$ je gramatika a nech δ a λ sú reťazce z $(N \cup \Sigma)^*$. Medzi reťazcami δ a λ platí binárna relácia $\Rightarrow_G$, nazývaná *derivačným krokom* (priamou deriváciou), ak sa môžu reťazce δ a λ vyjadriť v tvare:

$$\delta = \beta A \gamma,$$

$$\lambda = \beta \alpha \gamma,$$

kde β a γ sú ľobovoľné reťazce z $(N \cup \Sigma)^*$ a $A \Rightarrow \alpha$ je nejaké prepisovacie pravidlo z množiny P [7].

Výslednú vetu tvorí gramatika používaním derivačných krokov. Je teda možné zadefinovať tranzitívny a reflexívny uzáver relácie derivačného kroku.

Definícia 2.2.3. Nech $G = (N, \Sigma, P, S)$ je gramatika a nech δ a λ sú reťazce z $(N \cup \Sigma)^*$. Medzi reťazcami δ a λ platí binárna relácia $\Rightarrow_G^+$, nazývaná *deriváciou* (tranzitývnym uzáverom relácie derivačného kroku), ak existuje postupnosť generujúcich krokov $\mu_{i-1} \Rightarrow_G \mu_i$ pre $i = 1, \ldots, n \in \mathbb{N}$ taká, že platí:

$$\delta = \mu_0 \Rightarrow_G \mu_1 \Rightarrow_G \ldots \Rightarrow_G \mu_{n-1} \Rightarrow_G \mu_n = \lambda.$$

Táto postupnosť sa nazýva deriváciou dĺžky n. Ak platí $\delta \Rightarrow_G^+ \lambda$, potom sa hovorí, že reťazec λ je možné generovať z reťazca δ v gramatike G. Symbol $\Rightarrow^n$ označuje n-tú mocninu relácie $\Rightarrow$ [7].

Definícia 2.2.4. Ako píšu autori v [7], ak v gramatike G platí pre reťazce δ a λ relácia $\delta \Rightarrow^+ \lambda$ alebo identita $\delta = \lambda$, potom sa píše $\delta \Rightarrow^* \lambda$. Relácia $\Rightarrow^*$ je *tranzitívnym a reflexívnym uzáverom* relácie derivačného kroku $\Rightarrow$.

Ak sa prejde pomocou postupnosti derivačných krokov k reťazcu, ktorý obsahuje iba terminálne symboly, potom už nejde aplikovať žiadne prepisovacie pravidlo a generovanie končí. Z tejto skutočnosti je odvodený názov množiny Σ ako množina *terminálnych* symbolov. Ako posledná definícia tejto kapitoly je uvedená definícia jazyka generovaného gramatikou.

Definícia 2.2.5. Nech $G = (N, \Sigma, P, S)$ je gramatika. Reťazec $\alpha \in (N \cup \Sigma)^*$ sa nazýva vetnou formou, ak platí $S \Rightarrow^* \alpha$, t.j. reťazec α je generovateľný z počiatočného symbolu S. Vetná forma, ktorá obsahuje iba terminálne symboly, sa nazýva *veta*. *Jazyk* $L(G)$, generovaný gramatikou G, je podľa [7] definovaný množinou všetkých viet

$$L(G) = \{w \mid S \Rightarrow^* w \wedge w \in \Sigma^*\}.$$

Kapitola 3

Gramatický systém

Gramatický systém je formálny model slúžiaci na popis jazyka. Podobne ako gramatiky dokáže popísať jazyk, tento raz nie však pravidlami, ale využitím svojich častí – komponentov. *Komponent gramatického systému* je takmer totožný s gramatikou definovanou v 2.1, avšak správaním sa mierne odlišuje. Komponenty gramatického systému dokážu spolu komunikovať, striedať sa na generovaní určitých častí jazyka a spoluprácou vytvoriť výslednú vetu. Táto forma komunikácie prebieha s pomocou *komunikačných symbolov*. Komunikačné symboly sú terminálne alebo neterminálne symboly, vďaka ktorým je schopný gramatický systém predať riadenie generovania z jednoho komponentu na druhý. Množina komunikačných symbolov K je zložená z prvkov, ktoré patria do množín terminálnych alebo neterminálnych symbolov jednotlivých komponentov. Formálne množinu K môžeme vyjadriť ako:

$$K \subseteq \bigcup_{i=1}^{n} (N_i \cup \Sigma_i). \tag{3.1}$$

Generovanie na samotnom začiatku môže začínať ľubovoľný komponent, ktorý začína od svojho počiatočného netreminálu. Gramatický systém teda môžeme nazvať *decentralizovaným*, čo znamená, že jednotlivé komponenty sú si medzi sebou rovné. Pri *centralizovanom* gramatickom systéme dochádza k zavedeniu hlavného, centrálneho komponentu, ktorý začína generovanie a v priebehu generovania sa komunikuje vždy s ním. Komponent, ktorý má v daný moment riadenie generovania vetnej formy na starosť nazývame *aktívnym*. V gramatickom systéme sa môže v jeden moment nachádzať len **jeden** aktívny komponent. Ostatné neaktívne komponenty čakajú, kým aktívny komponent nenarazí pri generovaní na komunikačný symbol a gramatický systém predá vedenie inému komponentu podľa funkcie f. Funkcia f mapuje komunikačný symbol na komponent gramatického systému a tým dokáže zaistiť, aby pri výskyte komunikačného symbolu vo vetnej forme bol gramatický systém schopný prepnúť kontext správnemu komponentu.

Definícia 3.0.1. *Gramatický systém* Γ je n-tica $\Gamma = (G_1, G_2, \ldots, G_n, K, f)$, kde

- G_i, $1 \leq i \leq n \in \mathbb{N}$ je komponent gramatického systému, pričom kardinalita množiny $\{G_1, \ldots, G_n\}$ je kladné, nenulové číslo,
- K je množina komunikačných symbolov, $K \subseteq \bigcup_{i=1}^{n} (N_i \cup \Sigma_i)$,
- $f : K \to \{G_1, \ldots, G_n\}$ je funkcia, ktorá mapuje komunikačný symbol na komponent gramatického systému Γ.

Konvencia 3.0.1. Pre účely tejto kapitoly zaveďme nasledujúce konvencie:

- nech $n \in \mathbb{N}$ značí počet komponentov gramatického systému Γ,
- nech $i, j \in \mathbb{N} \wedge 1 \leq i \neq j \leq n$ označujú číslo komponentu gramatického systému Γ,
- nech $G_i = (N_i, \Sigma_i, P_i, S_i), G_j = (N_j, \Sigma_j, P_j, S_j)$ sú komponenty gramatického systému Γ.

Praktickú aplikáciu gramatických systémov je možné vidieť v syntaktickej analýze, v ktorej sa vstupný reťazec číta zľava doprava. Gramatické systémy pracujú tiež striktne zľava doprava, čo značne uľahčuje prácu s komunikačnými symbolmi. Komunikačný symbol, ktorý je zároveň terminálom, totiž z generovanej vetnej formy nemizne, tým pádom by nebolo možné určiť, či sa už využil na prepnutie kontextu. Zavedením obmedzenia pohybu sa už k „použitému“ komunikačnému symbolu gramatický systém nevráti, tým pádom nie je nutné tento probém ošetrovať.

Najdôležitejšou časťou gramatického systému je jeho komponent. Rovnako ako gramatiky využíva množiny terminálnych a neterminálnych symbolov. Množina terminálnych symbolov sa v tomto prípade rovná len podmnožine abecedy, nad ktorou je definovaný výstupný jazyk gramatického systému, pokiaľ sa v gramatickom systéme nenachádza len jeden komponent. V tomto prípade je vyjadrovacia sila gramatického systému zhodná s bezkontextovými gramatikami.

Definícia 3.0.2. V gramatickom systéme Γ je *komponent* G_i štvorica $G_i = (N_i, \Sigma_i, P_i, S_i)$, $1 \leq i \leq n$, kde jeho jednotlivé prvky sú zhodné s prvkami bezkontextovej gramatiky definovanej v Definícii 2.2.1. Komponenty môžu nadobúdať aj tvar kontextových gramatík, kedy je množina ich pravidiel v tvare zo Zápisu 2.3.

Gramatický systém je možné prehlásiť ako všeobecný nástroj, ktorého komponenty môžu byť teoreticky ľubovoľnými gramatikami. Typ výsledného jazyka je závislý na type komponentov gramatického systému. Práca sa zaoberá len bezkontextovými a kontextovými gramatikami ako komponentmi gramatického systému.

Aplikáciou prepisovacieho pravidla na vetnú formu vznikne nová vetná forma alebo sa zmení aktívny komponent. Táto zmena sa pri komponente označuje rovnako ako pri gramatike názvom *derivačný krok*. Gramatický systém narozdiel od gramatík vykonáva až dva druhy derivačných krokov:

- generujúci krok,
- krok zmeny komponentu.

Kroky zmeny komponentov majú v gramatickom systéme vyššiu prioritu ako generujúce kroky.

Generujúci krok je derivačný krok gramatického systému, ktorý dokáže aplikovať prepisovacie pravidlá v rámci jednoho komponentu. Definíciou a chovaním je podobný derivačnému kroku pri gramatikách. Gramatický systém vykoná generujúci krok ${}_g\Rightarrow$ práve vtedy, ak aktuálne spracovavaný symbol nepatrí do množiny komunikačných symbolov K. Ak platí medzi reťazcami δ a λ relácia generujúceho kroku, potom sa píše $\delta \ {}_g\Rightarrow_{G_i} \ \lambda$ a hovorí, že reťazec λ dokážeme priamo generovať z reťazca δ pomocou komponentu G_i.

Definícia 3.0.3. Nech δ a λ sú reťazce z $(N_i \cup \Sigma_i)^*$ v aktívnom komponente G_i. Medzi reťazcami δ a λ platí binárna relácia ${}_g\Rightarrow_{G_i}$, nazývaná *generujúci krok*, ak sa môžu reťazce δ a λ vyjadriť v tvare:

$$\delta = \beta A \gamma,$$
$$\lambda = \beta \alpha \gamma,$$

kde β a γ sú ľubovoľné reťazce z $(N_i \cup \Sigma_i)^*$ a $A \to \alpha$ je nejaké prepisovacie pravidlo z množiny P_i.

Charakteristickým chovaním gramatického systému je zmena aktívneho komponentu. Krok zmeny komponentu umožňuje túto zmenu vykonať. Pomocou komunikačného symbolu z množiny K a funkcie f dokáže zistiť, ktorý komponent sa stane aktívnym. Neformálne popísané, ak sa v generovanej vetnej forme nenachádza neterminálny symbol z množiny neterminálnych symbolov komponentu na ktorý sa prepína, tak sa začína od počiatočného symbolu tohto komponentu. Ak sa tam aspoň jeden takýto neterminálny symbol nachádza, komponent začína od prvého takého. Zavedením prvej situácie rieši problém, keby gramatický systém predá vedenie komponentu, ktorý nemá odkiaľ začínať. V tomto prípade začína od svojho počiatočného neterminálu. Táto vlastnosť gramatického systému je kľúčová, keďže používa viacero gramatík, ktoré spolu menia a generujú jednu výslednú vetu. Prevedením tohoto derivačného kroku sa gramatický systém už nikdy nevráti ku komunikačnému symbolu, pomocou ktorého bol krok zmeny komponentu prevedený z dôvodu, že spracováva symboly zľava doprava. Keby sa gramatický systém dostal ku komunikačnému symbolu, ktorý sa podľa funkcie f mapuje na aktívny komponent, krok zmeny komponentu sa nevykoná.

Definícia 3.0.4. Nech δ a λ sú reťazce z $(N_i \cup \Sigma_i \cup K \cup N_j \cup \Sigma_j)^*$ z G_i, G_j a nech je G_i aktívnym komponentom. Medzi reťazcami δ a λ platí binárna relácia ${}_c\Rightarrow_{G_j}$, nazývaná *krok zmeny komponentu*, ak sa môžu reťazce δ a λ vyjadriť buď v tvare

$$\delta = \lambda = \alpha k \gamma \beta,$$

alebo v tvare

$$\delta = \alpha k \beta,$$
$$\lambda = \alpha k S_j \beta,$$

kde α a β sú ľubovoľné reťazce z $(N_i \cup \Sigma_i \cup \Sigma_j)^*$, γ je reťazec z $(N_j \cup \Sigma_j)^* N_j (N_j \cup \Sigma_j)^*$, k je komunikačný symbol z množiny K, $f(k) = G_j$ a S_j je počiatočný symbol komponentu G_j. Komponent G_i predáva aktivitu komponentu G_j pričom prvá forma má prednosť pred druhou.

Pri gramatikách má zmysel definovať reflexívny a tranzitívny uzáver derivačného kroku, avšak pri gramatickom systéme to má význam len pri kroku generujúcom. Krok zmeny komponentu slúži výhradne len na zmenu aktívneho komponentu. Ak sa vylúči prípad umelého vygenerovania počiatočného symbolu z dôvodu uvedeného vyššie, tak krok zmeny komponentu negeneruje žiadnu vetnú formu a komunikačný symbol sa spotrebuje. Tým pádom nie je možné vykonať dva kroky zmeny komponentu bezprostredne po sebe. Nasleduje definícia tranzitívneho a reflexívneho uzáveru generujúceho kroku, ktorý je podobný s definíciou pri gramatikách.

Definícia 3.0.5. Nech δ a λ sú reťazce z $(N_i \cup \Sigma_i)^*$ a aktívny komponent G_i. Medzi reťazcami δ a λ existuje binárna relácia ${}_g\Rightarrow^+_{G_i}$, nazývaná *tranzitívny uzáver generujúceho kroku*, ak existuje postupnosť generujúcich krokov $\mu_{i-1}\ {}_g\Rightarrow_{G_i}\ \mu_i$ pre $i = 1, \ldots, n \in \mathbb{N}$ taká, že platí:

$$\delta = \mu_0\ {}_g\Rightarrow_{G_i}\ \mu_1\ {}_g\Rightarrow_{G_i}\ \cdots\ {}_g\Rightarrow_{G_i}\ \mu_{n-1}\ {}_g\Rightarrow_{G_i}\ \mu_n = \lambda.$$

Táto postupnosť sa nazýva aj deriváciou generujúceho kroku dĺžky n. Ak platí $\delta\ {}_g\Rightarrow^+_{G_i}\ \lambda$, potom sa hovorí, že reťazec λ sa môže generovať z reťazca δ pomocou komponentu G_i. Symbolom ${}_g\Rightarrow^n$ sa označuje n-tú mocnina relácie ${}_g\Rightarrow$.

Definícia 3.0.6. Reláciu ${}_g\Rightarrow^*$ nazývame *tranzitívnym a reflexívnym uzáverom* relácie ${}_g\Rightarrow$ ak platí

$$\delta\ {}_g\Rightarrow^+\ \lambda, \text{ alebo } \delta = \lambda.$$

Definícia jazyka prijímaného gramatickým systémom sa odlišuje od jazyka prijímaného pomocou gramatík, z dôvodu že môže vykonávať dva druhy derivačných krokov v rôznom množstve a poradí.

Definícia 3.0.7. Nech S_i, S_j sú počiatočné symboly komponentov G_i, G_j. *Jazyk* $L(\Gamma)$, generovaný gramatickým systémém Γ je množina reťazcov $w \in \Sigma^*$ generovaných deriváciou:

$$\begin{aligned} S_{i_1}\ {}_g\Rightarrow^*_{G_{i_1}}\ & \alpha k_{j_1}\beta\ {}_c\Rightarrow_{G_{j_1}}\ \alpha k_{j_1} S_{j_1}\beta \ldots \\ {}_g\Rightarrow^*_{G_{i_2}}\ & \alpha k_{j_2}\beta\ {}_c\Rightarrow_{G_{j_2}}\ \alpha k_{j_2} S_{j_2}\beta \ldots \\ & \vdots \\ {}_g\Rightarrow^*_{G_{i_h}}\ & w, \end{aligned}$$

pre nejaké $1 \leq m \leq h, i_m \in \{1, \ldots n\}$.

3.1 Príklady fungovania gramatického systému

Táto sekcia rozoberá praktické príklady fungovania gramatického systému pri generovaní bezkontextového a kontextového jazyka.

Konvencia 3.1.1. V každom príklade v tejto sekcii uvažujme o gramatickom systéme v jeho decentralizovanom variante. Bez ujmy na všeobecnosti ďalej začínajme generovanie v prvom komponente gramatického systému.

Príklad 3.1.1. Zadaný je bezkontextový jazyk L_1:

$$L_1 = \{a^n b^n \mid n \geq 1\}. \tag{3.2}$$

Na generovanie tohoto jazyka je vytvorený gramatický systém s dvomi komponentmi $\Gamma_1 = (G_1, G_2, K, f)$, kde:

$$G_1 = (\{S, A, B\}, \{a\}, P_1, S), \tag{3.3}$$

$$P_1 = \left\{ \begin{array}{ll} S \to AB, & A \to aAB, \\ A \to a & \end{array} \right\}, \tag{3.4}$$

$$G_2 = (\{B\}, \{b\}, \{B \to b\}, B), \tag{3.5}$$

$$K = \{B\}. \tag{3.6}$$

Mapovacia funkcia f nech v tomto príklade vyzerá nasledovne:

K	$\mathbf{G_i}$
B	G_2

Teda funkcia f mapuje jediný komunikačný neterminálny symbol B na komponent G_2. Je zrejmé, že jazyk L_1 obsahuje vety, ktoré majú súhlasný počet symbolov a a b. Generovanie týchto viet začína v komponente G_1, ktorá postupne zaisťuje, aby sa počet a a A rovnal počtu B. Svoje generovanie ukončí použitím pravidla $A \to a$, kde sa následne gramatický systém dostane k neterminálu B, ktorý symbolizuje komunikačný symbol, a gramatický systém predá vedenie komponentu G_2. Druhý komponent potom len prepíše neterminály B na terminály a ukončí generovanie. Uvedené sú dva príklady postupu derivačných krokov gramatického systému. Najprv pre vetu ab a potom pre a^3b^3. Podčiarknutý symbol symbolizuje výskyt komunikačného symbolu pred prevedením kroku zmeny komponentu.

$$S \;{}_g\!\Rightarrow_{G_1}\; AB \;{}_g\!\Rightarrow_{G_1}\; a\underline{B} \;{}_c\!\Rightarrow_{G_2}\; aB \;{}_g\!\Rightarrow_{G_2}\; ab$$
$$S \;{}_g\!\Rightarrow_{G_1}\; AB \;{}_g\!\Rightarrow_{G_1}\; aABB \;{}_g\!\Rightarrow_{G_1}\; aaABBB \;{}_g\!\Rightarrow_{G_1}\; aaa\underline{B}BB \;{}_c\!\Rightarrow_{G_2}\; aaaBBB \;{}_g\!\Rightarrow^{3}_{G_2}\; aaabbb$$

Pomocou gramatického systému je možné generovať aj jazyky kontextové, ktoré nie sú bezkontextové. Aby mohol gramatický systém takéto jazyky generovať, je potrebné, aby aspoň jeden komponent bol kontextovou gramatikou, teda mať svoju množinu pravidiel v tvare ako v Zápise 2.3. Nasledujúci príklad znázorňuje zladenú prácu troch komponentov s demonštráciou prepínania komponentov pomocou kroku zmeny komponentu.

Príklad 3.1.2. Zadaný je kontextový jazyk L_2:

$$L_2 = \{a^n b^n c^n \mid n \geq 1\}. \tag{3.7}$$

K nemu je definovaný gramatický systém s jedným bezkontextovým a dvoma nebezkontextovými komponentmi $\Gamma_2 = (G_1, G_2, G_3, K, f)$, kde:

$$G_1 = (\{S, A, B, C\}, \{a\}, P_1, S), \tag{3.8}$$
$$P_1 = \left\{ \begin{array}{ll} S \to abc, & S \to A, \\ A \to aBC, & A \to aABC \end{array} \right\}, \tag{3.9}$$
$$G_2 = (\{S_2, B, C, X, Y\}, \emptyset, P_2, S_2), \tag{3.10}$$
$$P_2 = \left\{ \begin{array}{ll} CB \to CX, & CX \to YX, \\ YX \to YC, & YC \to BC \end{array} \right\}, \tag{3.11}$$
$$G_3 = (\{S_3, B, C\}, \{a, b, c\}, P_3, S_3), \tag{3.12}$$
$$P_3 = \left\{ \begin{array}{ll} aB \to ab, & bB \to bb, \\ bC \to bc, & cC \to cc \end{array} \right\}, \tag{3.13}$$
$$K = \{B, C\}. \tag{3.14}$$

Mapovacia funkcia f nech v tomto príklade vyzerá nasledovne:

K	$\mathbf{G_i}$
B	G_3
C	G_2

Prvý komponent gramatického systému generuje súhlasný počet a, B a C. Pravidlom $S \to abc$ je zahrnutý prípad, kedy sa v definícii jazyka n rovná 1. Keby tento prípad nebol ošetrený týmto pravidlom, došlo by k nemožnosti generovania tejto vety, pretože by došlo k zmene aktívneho komponentu na G_2. Ten by potom nemohol vykonávať žiadne pravidlá, ani predávať vedenie, z dôvodu, že gramatický systém pracuje zľava doprava a komunikačný symbol C sa nachádza až na konci vetnej formy. Druhý komponent má na starosť výmenu výskytu reťazca CB za BC. Je vhodné poznamenať, že množina neterminálov obsahuje počiatočný symbol gramatiky S_2, ktorý nie je použitý. Tento symbol je v množine neterminálov, len z dôvodu Definície gramatiky 2.2.1 a nie je použitý, pretože sa využíva vlastnosť kroku zmeny komponentu, ktorá tento symbol generuje len v prípade, keď neexistuje výskyt neterminálov patriacich k danej gramatike ako je uvedené v Definícii 3.0.4. Tretí komponent prepisuje vhodné neterminály na terminály pomocou kontextových pravidiel. Podobne ako pri druhom sa z rovnakého dôvodu jeho počiatočný symbol nevyužíva.

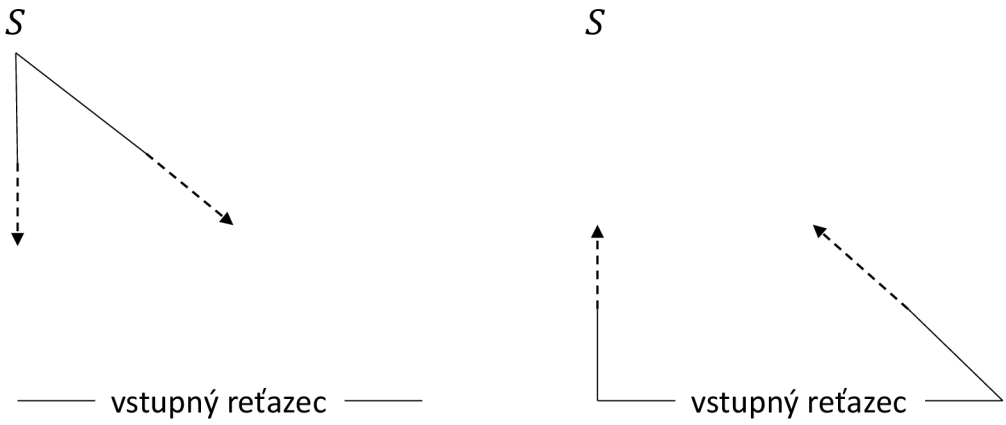
Pre tento gramatický systém je uvedená jedna derivačná postupnosť, ktorá vedie k vygenerovaniu vety $a^3b^3c^3$. Komunikačné symboly pred prevedením kroku zmeny komponentu sú podčiarknuté a pri generujúcom kroku je znázornená aplikácia generujúceho pravidla tučným písmom.

$$S \underset{g}{\Rightarrow}_{G_1} \mathbf{A} \underset{g}{\Rightarrow}_{G_1} \mathbf{aABC} \underset{g}{\Rightarrow}_{G_1} a\mathbf{aABC}BC \underset{g}{\Rightarrow}_{G_1} aa\mathbf{a}\underline{\mathbf{B}}\mathbf{C}BCBC \underset{c}{\Rightarrow}_{G_3} aaaBCBCBC \underset{g}{\Rightarrow}_{G_3} \underset{g}{\Rightarrow}_{G_3} aa\mathbf{ab}\underline{C}BCBC \underset{c}{\Rightarrow}_{G_2} aaabCBCBC \underset{g}{\Rightarrow}_{G_2} aaab\underline{\mathbf{B}}\mathbf{C}CBC \underset{c}{\Rightarrow}_{G_3} aaabBCCBC \underset{g}{\Rightarrow}_{G_3} \underset{g}{\Rightarrow}_{G_3} aaa\mathbf{bb}\underline{C}CBC \underset{c}{\Rightarrow}_{G_2} aaabbCCBC \underset{g}{\Rightarrow}_{G_2} aaabbC\mathbf{BC}C \underset{g}{\Rightarrow}_{G_2} aaabb\underline{\mathbf{B}}\mathbf{C}CC \underset{c}{\Rightarrow}_{G_3} \underset{c}{\Rightarrow}_{G_3} aaabbBCCC \underset{g}{\Rightarrow}^{4}_{G_3} aaabb\mathbf{bccc}$$

Kapitola 4

Syntaktická analýza

Syntaktická analýza v informatike označuje proces overovania syntaxe určitého objektu, typicky počítačového programu. Model, ktorý vykonáva syntaktickú analýzu sa nazýva *syntaktický analyzátor*, ktorý pracuje s prvkami, ktoré sa nazývajú *tokeny*. Tokeny sú, neformálne nazvané, slová (lexémy) vstupného programu (reťazca), ktoré reprezentujú najmenší syntaktický prvok. Skladaním týchto prvkov vzniká syntaktická štruktúra, skladaním štruktúr vzniká program. Syntaktický analyzátor k tomu, aby mohol pracovať s tokenmi, potrebuje lexikálny analyzátor, ktorý z lexém vytvára tokeny, s ktorými už dokáže syntaktický analyzátor ďalej pracovať. Postupnosť tokenov je overovaná gramatikou, ktorá je neoddeliteľnou súčasťou syntaktického analyzátora. Aplikovaním pravidiel gramatiky dokáže syntaktický analyzátor kontrolovať správnosť syntaxe vstupného programu a tým zaručiť správnosť zápisu. Prirodzene človeku, syntaktický analyzátor číta túto postupnosť tokenov zľava doprava. Syntaktická analýza sa rozdeľuje na dve hlavné časti: na syntaktickú analýzu zhora nadol a zdola nahor. Toto pomenovanie vychádza z tvorenia *syntaktického stromu*, ktorý reprezentuje štruktúru programu podľa aplikovaných pravidiel gramatiky. Postup tvorenia tohto stromu odlišuje jednu metódu od druhej. Pričom metóda syntaktickej analýzy zhora nadol postupuje od koreňa syntaktického stromu k listom. Metóda syntaktickej analýzy zdola nahor pracuje opačne. Obe metódy sú znázornené na Obrázku 4.1.



(a) Syntaktická analýza zhora nadol.

(b) Syntaktická analýza zdola nahor.

Obr. 4.1: Metódy syntaktickej analýzy [4].

Koreň v tomto syntaktickom strome reprezentuje v gramatike počiatočný symbol a listy predstavujú terminály alebo konkrétnejšie jednotlivé tokeny overovaného programu. Najdôležitejšou požadovanou vlastnosťou syntaktického analyzátora je determinizmus. V praxi je očakávané od syntaktického analyzátora, že bude pracovať deterministicky, to znamená, v každom momente má vedieť, aký krok má vykonať. Obsahom kapitoly je predstavenie metód analýzy zhora nadol a zdola nahor. K týmto metódam sú uvedené postupy, algoritmy a dátové štruktúry, s ktorými tieto metódy pracujú. Na konci kapitoly je predstavený gramatický systém z Kapitoly 3 v aplikácii v syntaktickej analýze.

4.1 Deterministická analýza zhora nadol

Metóda syntaktickej analýzy zhora nadol funguje simulovaním zostrojenia syntaktického stromu vstupného programu. Rovnako ako metóda zdola nahor spracováva vstupnú postupnosť tokenov obdržanú od lexikálneho analyzátora zľava doprava, ale vertikálne v inom poradí, a to zhora nadol. Gramatika, z ktorej vychádza syntaktický analyzátor, sa v metóde zhora nadol nazýva LL gramatika, kde prvé L označuje anglicky *left-to-right* (zľava doprava) overovanie tokenov a druhé L označuje anglicky *leftmost* (najľavejšiu) deriváciu, ktorú syntaktický analyzátor vykonáva. Táto LL gramatika musí byť dopredu zostrojená a prevedená z klasickej bezkontextovej gramatiky na LL gramatiku úpravou svojich prepisovacích pravidiel. Niektoré literárne zdroje píšu o LL gramatikách detailnejšie ako o *LL(1) gramatikách*. Číslo 1 v tomto prípade znamená jeden token, na ktorý sa syntaktický analyzátor pozerá pri prechádzaní vstupného reťazca počas jedného kroku. V skutočnosti sú LL(1) gramatiky špeciálnym prípadom *LL(k) gramatík*, kde $k \geq 1$, ktoré obdobne nahliadajú na k vstupných tokenov dopredu. Vo zvyšku práce hovorme pre sprehľadnenie o LL gramatikách(1) ako o LL gramatikách, pokiaľ nebude napísané inak.

Ďalším pomocníkom syntaktického analyzátora sú dopredu zostrojené matematické množiny, vďaka ktorým si vie syntaktický analyzátor zabezpečiť svoju deterministickosť. Tieto množiny sú zostrojené algoritmami, ktoré pracujú s gramatikou a jej pravidlami, prípadne inými množinami a sú detailnejšie popísané na začiatku sekcie. Nasleduje konštrukcia LL tabuľky z LL gramatiky a na konci je uvedený popis metódy rekurzívneho zostupu.

Množina predict a LL gramatiky

Ako bolo na začiatku tejto sekcie spomenuté, tak na to, aby mohol byť proces syntaktickej analýzy v praxi správne vykonávaný, je potrebné zabezpečiť deterministickosť syntaktického analyzátora v každom prípade. Jedným z príkladov, kedy syntaktický analyzátor využije túto vlastnosť je, keď existuje viacero pravidiel s rovnakou ľavou stranou, ale syntaktický analyzátor potrebuje vedieť, ktoré z nich vybrať. Krok späť nie je v syntaktickej analýze možný, tak nesprávny krok by mohol znamenať neprijatie vstupného reťazca, aj keby bol syntakticky v správnej forme. Rozhodnutie, ktoré analyzátor vykoná, je riadené množinou $Predict$, ktorá obsahuje pre každé pravidlo určité terminálne symboly. Tieto terminálne symboly sú symbolmi, ktorými môže počínať reťazec vzniknutý z derivácie, ktorej prvý krok vykoná toto pravidlo. Ak dve pravidlá majú svoje množiny $Predict$ disjunktné, tak si syntaktický analyzátor dokáže podľa tokenu, ktorý je aktuálne na vstupe, vybrať správne pravidlo, a tak viesť postup derivácie správnym smerom. Na zostrojenie množiny $Predict$, je potrebné najprv zadefinovať množiny $Empty$, $First$ a $Follow$.

Definícia 4.1.1. Nech $G = (N, \Sigma, P, S)$ je gramatika, a $\alpha \in (N \cup \Sigma)^*$. Množina $Empty(\alpha)$ je definovaná ako $Empty(\alpha) = \varepsilon$, ak platí $\alpha \Rightarrow_G^* \varepsilon$, inak $Empty(\alpha) = \emptyset$ [5].

Ak gramatiky dokáže reťazec úplne vymazať pomocou ε-pravidiel, tak množina *Empty* bude pre tento reťazec obsahovať prvok ε, čo bude symbolizovať možnosť vymazania reťazca. Nasleduje algoritmus na výpočet množiny *Empty* [3].

Algoritmus 4.1.1 Množina $Empty(X)$

Vstup: gramatika $G = (N, \Sigma, P, S)$
Výstup: množina $Empty(X)$, pre každý symbol $X \in N \cup \Sigma$

```
1: ∀a ∈ Σ : Empty(a) ← ∅
2: for all A ∈ N do
3:     if A → ε ∈ P then
4:         Empty(A) ← {ε}
5:     else
6:         Empty(A) ← ∅
7:     end if
8: end for
9: while je možné meniť nejakú množinu Empty(X) do
10:    if A → X1X2...Xn ∈ P and ∀i = 1,...,n ∈ ℕ : Empty(Xi) = {ε} then
11:        Empty(A) ← {ε}
12:    end if
13: end while
```

Algoritmus na výpočet množiny *Empty* odohráva dôležitú úlohu v celkovom procese výpočtu ostatných množín. Na začiatku pre každý terminálny symbol určí jeho množinu *Empty* za prázdnu. Inými slovami, každý terminálny symbol nie je možné vymazať. Nasleduje určenie množiny *Empty* každého neterminálneho symbolu, ktorý sa nachádza na ľavej strane pravidla a na jeho pravej strane má prázdny reťazec za $\{\varepsilon\}$, teda že sa dá prepísať na prázdny reťazec. Ďalej nasleduje prechádzanie všetkých množín *Empty* a pozerá sa, či sa pravá strana pravidla nedá poskladať z neterminálov, ktorých množina *Empty* je neprázdna. Ak takéto pravidlo existuje, tak sa dá neterminál vymazať a množina *Empty* jeho pravej strany sa určí tiež za $\{\varepsilon\}$. Nutné je uviesť algoritmus pre výpočet množiny *Empty* ľubovoľného reťazca, ktorý sa používa v algoritmoch pre výpočet množín *First* a *Follow*.

Algoritmus 4.1.2 Množina $Empty(X_1X_2 \dots X_n)$

Vstup: gramatika $G = (N, \Sigma, P, S)$, $Empty(X)$ pre všetky symboly $X \in N \cup \Sigma$
Výstup: množina $Empty(X_1X_2 \dots X_n)$, kde $(X_1X_2 \dots X_n) \in (N \cup \Sigma)^+$

```
1: if ∀i = 1,...,n : Empty(Xi) = {ε} then
2:     Empty(X1X2...Xn) ← {ε}
3: else
4:     Empty(X1X2...Xn) ← ∅
5: end if
```

Množina $First(\alpha)$ je množina všetkých terminálnych symbolov, ktorými môže začínať reťazec derivovateľný z α.

Definícia 4.1.2. Nech $G = (N, \Sigma, P, S)$ je gramatika. Pre každý reťazec $\alpha \in (N \cup \Sigma)^*$ je v [5] definovaná množina $First(\alpha)$ ako:

$$First(\alpha) = \{a \mid a \in \Sigma, \alpha \Rightarrow_G^* a\beta, \beta \in (N \cup \Sigma)^*\}.$$

Algoritmus 4.1.3 Množina $First(X)$

Vstup: gramatika $G = (N, \Sigma, P, S)$
Výstup: množina $First(X)$, pre každý symbol $X \in N \cup \Sigma$

```
1: ∀a ∈ Σ : First(a) ← {a}
2: ∀A ∈ N : First(A) ← ∅
3: while je možné meniť nejakú množinu First(X) do
4:     if A → X_1X_2...X_{k-1}X_k...X_n ∈ P then
5:         First(A) ← First(A) ∪ First(X_1)
6:         if Empty(X_1X_2...X_{k-1}) = {ε} then
7:             First(A) ← First(A) ∪ First(X_k)
8:         end if
9:     end if
10: end while
```

Fungovanie algoritmu na výpočet množiny $First$ je priamočiare. Na začiatku algoritmus nastaví hodnoty terminálnych a neterminálnych symbolov a potom nasleduje prechádzaním pravidiel gramatiky. Množinu $First$ každého pravidla nastaví na $First$ jeho prvého symbolu a množinu $First$ všetkých symbolov, ktoré majú naľavo od seba vymazateľný jeden alebo viac symbolov. Pre potreby algoritmu $Follow$ a pre úplnosť je potrebné uviesť algoritmus na výpočet množiny $First$ pre neprázdne reťazce.

Algoritmus 4.1.4 Množina $First(X_1X_2 \dots X_n)$

Vstup: gramatika $G = (N, \Sigma, P, S)$, $Empty(X), First(X)$ pre všetky symboly $X \in N \cup \Sigma$
Výstup: množina $First(X_1X_2 \dots X_n)$, kde $(X_1X_2 \dots X_n) \in (N \cup \Sigma)^+$

```
1: First(X_1X_2...X_n) ← First(X_1)
2: while je možné meniť nejakú množinu First(X_1X_2...X_{k-1}X_k...X_n) do
3:     if Empty(X_1X_2...X_{k-1}) = {ε} then
4:         First(X_1X_2...X_n) ← First(X_1X_2...X_n) ∪ First(X_k)
5:     end if
6: end while
```

Ďalšou dôležitou informáciou, ktorou musí syntaktický analyzátor disponovať, je znalosť množiny terminálnych symbolov, ktoré sa môžu nachádzať za neterminálom A. Keby sa neterminál A dokázal prepísať v konečnom počte krokov na prázdny reťazec, tak analyzátor potrebuje poznať aké terminálne symboly môže očakávať za takýmto neterminálom. Práve na toto slúži množina $Follow$. Špeciálny symbol $\$ \notin \Sigma$ bude množina obsahovať pre neterminál, ktorý môže končiť vetnú formu.

Definícia 4.1.3. Nech $G = (N, \Sigma, P, S)$ je gramatika. Pre všetky $A \in N$ definujeme množinu $Follow(A)$ ako v [5] nasledovne:

$$Follow(A) = \{a \mid a \in \Sigma, S \Rightarrow_G^* \alpha A a \beta, \alpha, \beta \in (N \cup \Sigma)^*\} \cup \{\$ \mid S \Rightarrow_G^* \alpha A, \alpha \in (N \cup \Sigma)^*\}.$$

Algoritmus 4.1.5 Množina $Follow(A)$

Vstup: gramatika $G = (N, \Sigma, P, S)$
Výstup: množina $Follow(A)$, pre každý symbol $A \in N$

```
1:  Follow(S) ← $
2:  while je možné meniť nejakú množinu Follow(A) do
3:      if A → αBβ ∈ P then
4:          if β ≠ ε then
5:              Follow(B) ← Follow(B) ∪ First(β)
6:          end if
7:          if Empty(β) = {ε} then
8:              Follow(B) ← Follow(B) ∪ First(A)
9:          end if
10:     end if
11: end while
```

Konečne je možné uviesť definíciu poslednej množiny, množiny *Predict*. $Predict(A \to \alpha)$ je množina všetkých terminálnych symbolov, ktoré môžu byť aktuálne najľavejšie vygenerované, ak pre ľubovoľnú vetnú formu použijeme pravidlo $A \to \alpha$.

Definícia 4.1.4. Nech $G = (N, \Sigma, P, S)$ je gramatika. Pre každé $A \to \alpha \in P$ definujeme množinu $Predict(A \to \alpha)$ ako v [5]:

- ak $Empty(\alpha) = \{\varepsilon\}$, tak $Predict(A \to \alpha) = First(\alpha) \cup Follow(A)$,
- ak $Empty(\alpha) = \emptyset$, tak $Predict(A \to \alpha) = First(\alpha)$.

Množina *Predict* sa určí takýmto spôsobom pre každé prepisovacie pravidlo v gramatike. Slúži na to, aby mohol syntaktický analyzátor deterministicky vybrať pravidlo, ktoré sa aplikuje. Nech sa syntaktický analyzátor rozhoduje medzi výberom dvoch pravidiel p, r a nech na vstupe má symbol a. Ak sa symbol a nachádza v množine $Predict(p)$, tak syntaktický analyzátor sa rozhodne vykonať pravidlo p a tým zabezpečiť správne smerovanie postupnosti derivačných krokov až k výslednému generovanému slovu. Táto myšlienka vedie k definícii LL gramatík.

Definícia 4.1.5. Gramatika $G = (N, \Sigma, P, S)$ sa podľa [4] nazýva *LL gramatikou*, ak pre každé $A \in N$ a dve ľubovoľné pravidlá $p, q \in P$ platí:

$$p \neq q \ \wedge \ Predict(p) \cap Predict(q) = \emptyset.$$

Všeobecne platí, že jazyky generované LL gramatikami sú podmnožinou jazykov generovaných bezkontextovými gramatikami. Je možné previesť niektoré bezkontextové gramatiky na ekvivalentné LL gramatiky pomocou nasledujúcich techník [5]:

- faktorizácia (vytýkanie),
- odstránenie ľavej rekurzie.

Myšlienka faktorizácie spočíva v tom, že ak existuje viac pravidiel s rovnakou ľavou stranou, ktorých pravá strana sa začína rovnakým terminálom, tak sa zvyšok pravej strany nahradí novým neterminálom, z ktorého vzniknú nové pravidlá.

V gramatike sa teda prepisovacie pravidlá v tvare

$$A \to a\beta_1, A \to a\beta_2, \ldots A \to a\beta_n \tag{4.1}$$

zmenia na tvar

$$A \to aB, B \to \beta_1, B \to \beta_2, \ldots B \to \beta_n, \tag{4.2}$$

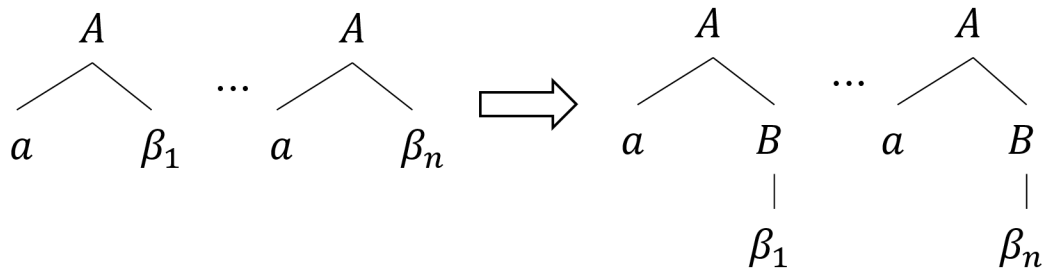
kde B je nový neterminál. Obrázok 4.2 tento proces znázorňuje. Odstránenie ľavej rekurzie prebieha zámenou pravidiel, ktoré sú v tvare ľavej rekurzie, t.j.

$$A \to A\beta, \ A \to x \tag{4.3}$$

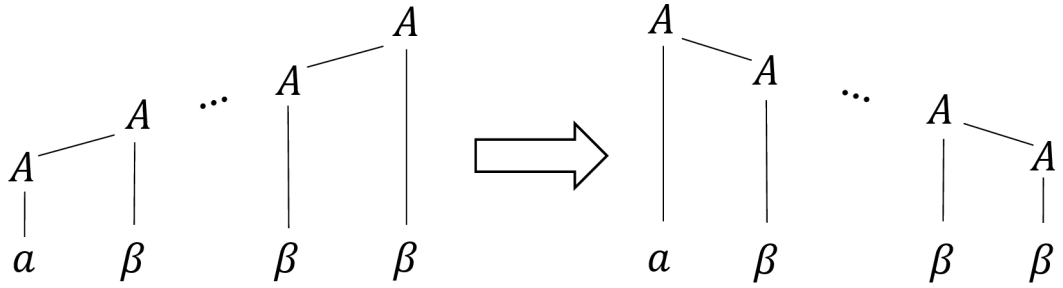
za pravidlá v tvare

$$A \to aB, \ B \to \beta B, \ B \to \varepsilon, \tag{4.4}$$

kde B je nový neterminál. Odstránenie ľavej rekurzie je znázornené na obrázku 4.3.



Obr. 4.2: Prevod pravidiel pomocou faktorizácie [5].



Obr. 4.3: Odstránenie ľavej rekurzie [5].

Konštrukcia LL tabuľky

LL tabuľka znázorňuje množinu *Predict* každého neterminálu LL gramatiky $G = (N, \Sigma, P, S)$. Riadky tabuľky reprezentujú neterminálne symboly z množiny N a stĺpce tabuľky terminálne symboly Σ spolu so špeciálnym symbolom \$, ktorý označuje koniec vstupného reťazca.

Hodnoty, ktoré tabuľka obsahuje, môžeme vyjadriť ako výstup z funkcie ρ, ktorá mapuje dvojicu (A, a), kde $A \in N, a \in \Sigma$ buď na pravidlo $r \in P$, ak $a \in Predict(r)$, pričom A je ľavou stranou pravidla r. Druhá možnosť je symbol $\times$, ktorý znázorňuje neexistenciu pravidla pre túto dvojicu, v tomto prípade teda $a \notin Predict(r)$. Formálne je funkcia ρ zapísaná v tvare:

$$\rho : (\Sigma \times N) \to \{P, \times\}. \tag{4.5}$$

Ak mohutnosť množiny $\rho(A, a)$ označíme k, tak, pre LL(k) gramatiky opísaných na začiatku Sekcie 4.2 existujú príslušné LL(k) tabuľky. Mohutnosť množiny $\rho(A, a)$ v Zápise 4.5 je 1, a teda LL tabuľku LL gramatiky G nazývame *LL(1) tabuľkou*. Vo zvyšku práce pre sprehľadnenie hovorme o LL(1) tabuľkách ako o LL tabuľkách.

Pravidlo r	$Predict(r)$
$1 : E \to TA$	$i, ($
$2 : A \to \vee\ TA$	$\vee$
$3 : A \to \varepsilon$	$), \$$
$4 : T \to FB$	$i, ($
$5 : B \to \wedge\ FB$	$\wedge$
$6 : B \to \varepsilon$	$\vee,), \$$
$7 : T \to F$	$i, ($
$8 : F \to (E)$	$($
$9 : F \to i$	i

Tabuľka 4.1: Pravidlá LL Gramatiky a ich množina *Predict*.

Tabuľka 4.1 zobrazuje príklad pravidiel a ich množiny *Predict* z [4]. Podľa postupu opísaného vyššie je možné skonštruovať LL tabuľku pre túto gramatiku.

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	$\times$	$\times$

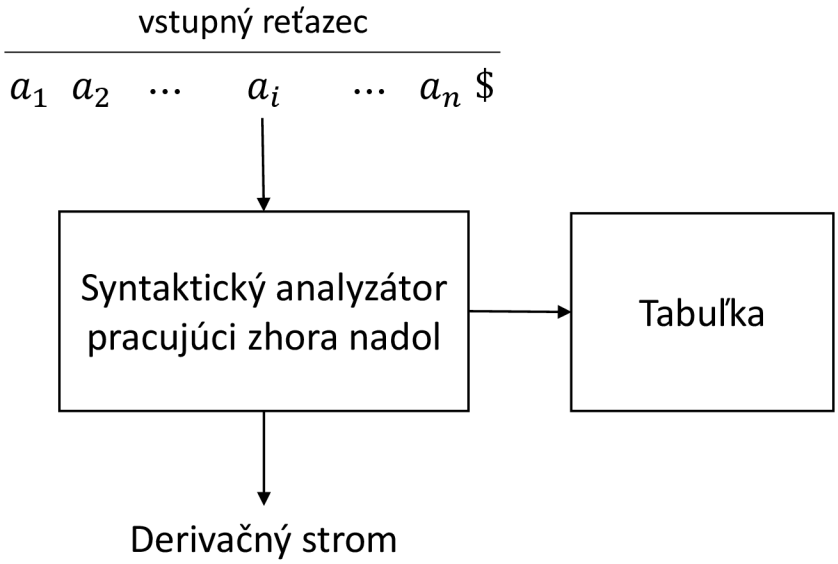
Tabuľka 4.2: LL tabuľka pre pravidlá z Tabuľky 4.1.

Túto tabuľku využívajú metódy syntaktickej analýzy pracujúce zhora nadol. Prvou je metóda prediktívneho rekurzívneho zostupu, ktorá využíva LL tabuľku ako pomoc pri impementácii programu. Druhou metódou využívajúcu LL tabuľku je metóda prediktívnej syntaktickej analýzy, ktorá spolu s LL tabuľkou úzko spolupracuje aj so zásobníkom. Práca sa ďalej zaoberá len metódou prediktívneho rekurzívneho zostupu, viac o prediktívnej syntaktickej analýze píše Meduna v [4] a autori v [1].

Preditkívny rekurzívny zostup

Hlavnou myšlienkou metódy prediktívneho rekurzívneho zostupu je simulácia zostrojenia derivačného stromu vstupného overovaného programu. V praxi, program, ktorý implemen-

tuje túto metódu má zostrojenú procedúru pre každý neterminálny symbol LL gramatiky, ktorá špecifikuje prijímaný jazyk. Ako jedna z metód analýz zhora nadol začína v počiatočnom symbole gramatiky, teda v procedúre, ktorá charakterizuje toto pravidlo. Následne v tele procedúry overuje vstupné tokeny, alebo pokračuje volaním ďalších procedúr. Analýza končí úspešným skončením počiatočnej procedúry a prečítaným vstupným reťazcom. Konštrukciu takýchto procedúr značne uľahčuje LL tabuľka, ktorá vychádza z LL gramatiky. Na základe množín *Predict* je jednoduchšie zostrojiť jednotlivé procedúry. Schéma syntaktického analyzátora pracujúceho touto metódou je zobrazená na Obrázku 4.4.



Obr. 4.4: Syntaktický analyzátor pracujúci zhora nadol.

Uvedený príklad lepšie vysvetľuje princíp rekurzívneho zostupu [4]. Nech sa syntaktický analyzátor riadi LL Tabuľkou 4.2 a uvažuje pravidlo

$$1: \ E \to TA, \tag{4.6}$$

ktorého množina

$$Predict(E \to TA) = \{i, (\}. \tag{4.7}$$

Potom v jeho implementácii bude procedúra, ktorá reprezentuje neterminál E vyzerať nasledovne:

```
function E: Boolean;
begin
  E := false;
  if token in {i, (} then
    begin
      if T then
        if A then E := true;
    end
end
```

kde `T` a `A` sú opäť metódy reprezentujúce neterminály T a A, `token` je vstupný token. Z LL tabuľky vyplíva, že ak sa na vstupe objaví symbol z množiny *Predict* pravidla 1, tak syntaktický analyzátor simuluje aplikáciu pravidla 4.6. Postupnosť volania týchto metód počas analýzy tvorí derivačný strom analyzovanej vety. Proces syntaktickej analýzy začína zavolaním procedúry pre počiatočný neterminál, ktorej výstup rozhodne o korektnosti zápisu vstupného reťazca:

```
if E then
  ACCEPT
else
  REJECT
```

Zotavenie syntaktického analyzátora pri narazení na chybu

Myšlienkou zotavenia z chyby je opätovný nález postupnosti tokenov, ktorú môže syntaktický analyzátor začať znovu overovať. Technika predstavená autormi v [1] popisuje zotavenie analyzátora pri metóde prediktívnej syntaktickej analýzy. Pri poupravení tejto techniky je možné jej ideu využiť aj v metóde rekurzívneho zostupu.

Ideou tejto techniky je preskakovanie vstupných tokenov do okamžiku, až sa objaví token z *množiny synchronizačných tokenov*. Syntaktický analyzátor potom pokračuje v analýze od tohoto tokenu. Efektivita zotavenia záleží na voľbe týchto synchronizačných tokenov. V praxi by tieto množiny mali obsahovať také tokeny, že sa syntaktický analyzátor dokáže rýchlo zotaviť z často urobených chýb. Obnovenie procesu syntaktickej analýzy v analyzátore fungujúcom na základe metódy rekurzívneho zostupu prebehne zavolaním procedúry, z ktorej je možné proces analýzy obnoviť. Jedným z problémov použitia tejto techniky pri metóde rekurzívneho zostupu je vynorenie sa z rekurzívneho zanorenia a nájsť procedúru, ktorá dokáže obnoviť analýzu. Problémy tohto druhu záležia na gramatike a implementácii metódy, preto sa riešenie ponecháva na programátora.

4.2 Deterministická analýza zdola nahor

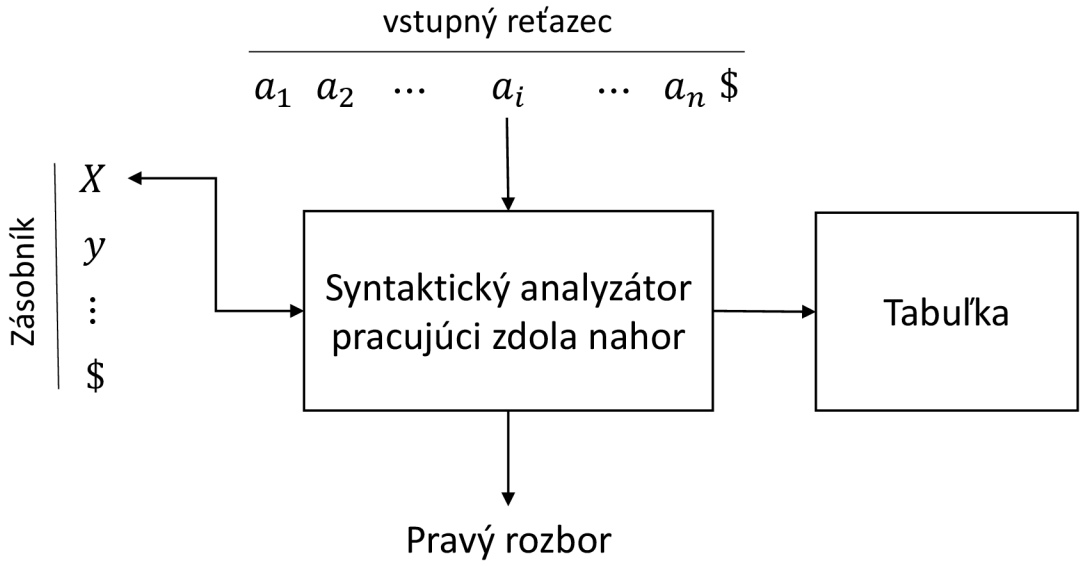
Syntaktický analyzátor pracujúci na základe metódy deterministickej syntaktickej analýzy zdola nahor overuje vstupný overovaný program. Rovnako ako druhá metóda zhora nadol, číta vstup zľava doprava, avšak postupuje opačne, a to od vstupného reťazca prevedeného na postupnosť tokenov až po koreň syntaktického stromu. V tomto prípade sa hovorí o najpravejšej derivácii. Syntaktický analyzátor pracujúci na základe tejto metódy vykonáva buď operáciu *shift*, alebo operáciu *redukcie*. Pri vykonaní operácie shift, syntaktický analyzátor presunie vstupný token na zásobník, s ktorým počas celej analýzy spolupracuje. Pri redukcii potrebuje syntaktický analyzátor na zásobníku nájsť *reťazec pripravený na redukciu* (anglicky *handle*). Tento reťazec sa následne zredukuje (prepíše) na ľavú stranu pravidla, ktorého pravá strana sa rovná tomuto reťazcu. Táto redukcia reprezentuje najpravejší derivačný krok. Podobne ako syntaktický analyzátor zhora nadol musí aj tento analyzátor pracovať deterministicky, aby mohol byť uplatnený v praxi.

Existujú dve základné deterministické metódy syntaktickej analýzy zdola nahor. Prvou, zložitejšou, metódou je *LR syntaktická analýza*, kde L označuje anglicky *left-to-right* (zľava doprava) overovanie vstupného reťazca a R označuje anglicky *rightmost* (najpravejšiu) deriváciu. LR syntaktické analyzátory sú rovnako silné ako deterministické zásobníkové automaty, tým pádom reprezentujú najsilnejší možný typ syntaktickej analýzy, ktorá fun-

guje deterministickým spôsobom. Viac o LR syntaktickej analýze sa dá dočítať v [4], táto sekcia sa zameria na populárnejšiu a jednoduchšiu metódu, a to *precedenčnú syntaktickú analýzu*, ktorá sa v praxi používa na kontrolu syntaxe výrazov, ktorých operátory a ich priority kontrolujú proces analýzy.

Algoritmus precedenčnej syntaktickej analýzy

Predpokladajme, že syntaktický analyzátor pracujúci metódou zdola nahor pracuje na základe gramatiky $G = (N, \Sigma, P, S)$. Tento syntaktický analyzátor pracuje pomocou svojej tabuľky precedencie. Riadky a stĺpce v tejto precedenčnej tabuľke sú tvorené prvkami z množiny $\Sigma \cup \{\$\}$. Hodnoty v precedenčnej tabuľke sú tvorené prvkami z množiny $\{>, <, =, \checkmark, \times\}$. Ďalšou neoddeliteľnou súčasťou syntaktického analyzátora je jeho zásobník. Ten využíva na ukladanie terminálnych a neterminálnych symbolov, spolu so špeciálnym znakom $<$, pomocou ktorého si určuje začiatok reťazca pripraveného na redukciu, dno zásobníka označuje symbolom \$. Teda jeho zásobníková abeceda je množina $\{\Sigma, <, \$\}$. Schéma takéhoto analyzátora je znázornená na Obrázku 4.5.



Obr. 4.5: Model pre syntaktickú analýzu pracujúcu zdola nahor [6].

Kroky syntaktického analyzátora určuje jeho tabuľka precedencie. Riadok v tejto tabuľke reprezentuje najvrchnejší terminálny symbol na zásobníku. Ak takýto symbol neexistuje, tak sa berie namiesto neho špeciálny symbol \$, ktorý algoritmus predpokladá, že nie je v množine terminálnych symbolov. Stĺpce tabuľky reprezentujú aktuálne spracovávaný symbol zo vstupného reťazca. Ak sa na priesečníku riadka a stĺpca nachádza symbol $<$, alebo $=$, tak syntaktický analyzátor vykoná operáciu shift. Pri symbole $<$ syntaktický analyzátor si pred presunutím vstupného symbolu presunie na zásobník aj symbol $<$, ktorý využije pri redukcii. Redukciu syntaktický analyzátor vykoná v momente, keď sa objaví $>$ v tabuľke precedencie a tá prebieha nasledovne. Syntaktický analyzátor určí výskyt reťazca α, kde $\alpha =< a\beta, a \in \Sigma, \beta \in (N \cup \Sigma)^*$. Reťazec α sa nazýva reťazec pripravený na redukciu (angl. handle). Po určení reťazca α, analyzátor vyberie pravidlo, ktorého pravá strana sa rovná reťazcu α a zredukuje (prepíše) reťazec α na svojom zásobníku na ľavú stranu vybraného pravidla. Symbol $\times$ reprezentuje v tabuľke syntaktickú chybu a sym-

bol ✓označuje úspešné dokončenie procesu syntaktickej analýzy. Algoritmus precedenčnej analýzy [4] používa operácie $REDUCE$ a $SHIFT$, ktoré je potrebné najprv zadefinovať.

Definícia 4.2.1. Syntaktický analyzátor na báze gramatiky $G = (N, \Sigma, P, S)$ pracujúci metódou zdola nahor používa operácie $REDUCE$ a $SHIFT$, ktoré modifikujú zásobník nasledujúcim spôsobom:

- $REDUCE(A \to \alpha)$, kde $A \in N, \alpha =< a\beta, a \in \Sigma, \beta \in (N \cup \Sigma)^*$ nahradí reťazec α na zásobníku za neterminál A,
- $SHIFT(<)$ presunie reťazec $\gamma =< a$, kde $a \in \Sigma$ je vstupný symbol na vrchol zásobníka a číta nasledujúci vstupný symbol,
- $SHIFT(=)$ presunie vstupný symbol $a \in Sigma$ na vrchol zásobníka a číta nasledujúci vstupný symbol [4].

Algoritmus 4.2.1 Precedenčný syntaktický analyzátor

Vstup: gramatika $G = (N, \Sigma, P, S)$, tabuľka precedencie, vstupný reťazec w ukončený symbolom \$, zásobník
Výstup: $ACCEPT$, ak $w \in L(G)$, $REJECT$, ak $w \notin L(G)$

```
1:  polož symbol $ na dno zásobníka
2:  repeat
3:      switch precedence_table[stack_top_terminal, input] do
4:          case = :
5:              SHIFT(=)
6:          case < :
7:              SHIFT(<)
8:          case > :
9:              if G obsahuje pravidlo A → α  and α = handle then
10:                 REDUCE(A → α)
11:             else
12:                 REJECT                          ▷ pravidlo neexistuje
13:             end if
14:         case × :
15:             REJECT                              ▷ tabuľka detekovala chybu
16:         case ✓ :
17:             ACCEPT
18: until ACCEPT  or REJECT
```

Ako bolo spomenuté, tak tento algoritmus sa v praxi využíva najmä na syntaktickú analýzu výrazov. Nasleduje ukážka fungovania algoritmu na príklade z [4]. Nech existuje gramatika $G = (\{E\}, \{\wedge, \vee, i, (,)\}, P, E)$ a precedenčná tabuľka znázornená v Tabuľke 4.3. Prepisovacie pravidlá P sú definované nasledovne:

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

K pravidlám bola dopredu skonštruovaná precedenčná tabuľka.

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

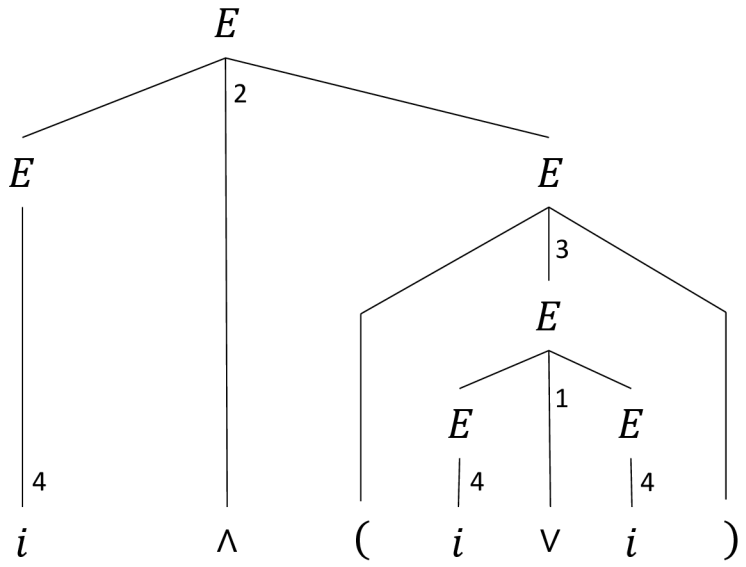
Tabuľka 4.3: Precedenčná tabuľka k pravidlám gramatiky G.

Vstupným reťazcom nech je $i \wedge (i \vee i)\$$. V Tabuľke 4.4 je znázornený postup precedenčnej syntaktickej analýzy s využitím precedenčnej tabuľky 4.3. Prvý stĺpec obsahuje konfiguráciu syntaktického analyzátora v tvare $\alpha \mid \beta$, kde $\alpha \in (\{\$\} \cup \{<\} \cup \Sigma)^*$ je zásobník používaný syntaktickým analyzátorom a $\beta \in (\{\$\} \cup \Sigma)^*$ je vstupným reťazcom. Druhý stĺpec obsahuje hodnoty v precedenčnej tabuľke 4.3 a tretí stĺpec obsahuje operáciu vykonanú syntaktickým analyzátorom. Podčiarknutý symbol označuje najvrchnejší terminálny symbol na zásobníku. Zjavne reťazec $i \wedge (i \vee i)$ patrí do jazyka generovaného gramatikou G, takže syntaktický analyzátor skončí úspešne.

Konfigurácia	Hodnota v tabuľke	Operácia
$\underline{\$} \mid i \wedge (i \vee i)\$$	$[\$, i] = <$	$SHIFT(<)$
$\$ < \underline{i} \mid i \wedge (i \vee i)\$$	$[i, \wedge] = >$	$REDUCE(4 : E \to i)$
$\underline{\$}\ E \mid\ \wedge (i \vee i)\$$	$[\$, \wedge] = <$	$SHIFT(<)$
$\$ < E \underline{\wedge} \mid (i \vee i)\$$	$[\wedge, (] = <$	$SHIFT(<)$
$\$ < E \wedge < \underline{(} \mid i \vee i)\$$	$[(, i] = <$	$SHIFT(<)$
$\$ < E \wedge < (< \underline{i} \mid\ \vee i)\$$	$[i, \vee] = >$	$REDUCE(4 : E \to i)$
$\$ < E \wedge < (E \mid i)\$$	$[(, \vee] = <$	$SHIFT(<)$
$\$ < E \wedge < (< E \underline{\vee} \mid i)\$$	$[\vee, i] = <$	$SHIFT(<)$
$\$ < E \wedge < (< E \vee < \underline{i} \mid)\$$	$[i,)] = >$	$REDUCE(4 : E \to i)$
$\$ < E \wedge < (< E \underline{\vee} E \mid)\$$	$[\vee,)] = >$	$REDUCE(1 : E \to E \vee E)$
$\$ < E \wedge < (E \mid)\$$	$[(,)] = =$	$SHIFT(=)$
$\$ < E \wedge < (E) \mid \$$	$[), \$] = >$	$REDUCE(3 : E \to (E))$
$\$ < E \underline{\wedge} E \mid \$$	$[\vee, \$] = >$	$REDUCE(2 : E \to E \wedge E)$
$\underline{\$}\ E \mid \$$	$[\$, \$] =$ ✓	$ACCEPT$

Tabuľka 4.4: Postup precedenčného syntaktického analyzátora.

Pravým rozborom vety je postupnosť čísiel 4 4 4 1 3 2, čo je vlastne obrátená postupnosť pravidiel použitých pri pravej derivácii tejto vety. Syntaktický strom vytvorený pre tento reťazec je zobrazený na Obrázku 4.6.



Obr. 4.6: Syntaktický strom výrazu $i \wedge (i \vee i)$.

Dôležitou súčasťou syntaktického analyzátora je jeho precedenčná tabuľka, ktorá udáva druh operácie, ktoré má syntaktický analyzátor vykonať v daný moment. Posledná časť sekcie je venovaná procesu zostrojenia tejto tabuľky.

Konštrukcia precedenčnej tabuľky

Tabuľku precedencie je nutné vyplniť ručne použitím základných matematických pravidiel. Predstavme konštrukciu tabuľky, ktorá je na báze gramatiky $G = (N, \Sigma, P, S)$. Dva terminály $a, b \in \Sigma$ sú v relácii $<, >, \times,$ alebo ✓ podľa pravidiel z [6]:

1. Precedencia operátorov:
   Ak je a operátorom s vyššou precedenciou ako operátor b, potom:

   $$a > b \text{ a } b < a.$$

2. Asociatívnosť:
   Ak sú a a b operátory asociatívne zľava, potom:

   $$a > b \text{ a } b > a.$$

   Ak sú a a b operátory asociatívne sprava, potom:

   $$a < b \text{ a } b < a.$$

3. Identifikátory:
   Ak a môže byť hneď pred operandom i, potom:

   $$a < i.$$

   Ak a môže byť hneď za operandom i, potom:

   $$i > a.$$

4. Zátvorky:
   Pre jeden pár zátvoriek platí: $(\doteq)$.
   Ak $a \in \Sigma - \{), \$\}$, potom: $(< a$.
   Ak $a \in \Sigma - \{(, \$\}$, potom: $a <)$.
   Ak a môže byť bezprostredne pred (, potom: $a <\ ($.
   Ak a môže byť bezprostredne za), potom: $)\ > a$.

5. Ukončovač reťazca \$:
   Nech a je ľubovoľný operátor, alebo operand, potom:

$$\$ < a \quad \text{a} \quad a > \$ \quad \text{a} \quad \$ \checkmark \$.$$

6. Zvyšok tabuľky vyplníme $\times$, čo označuje syntaktickú chybu.

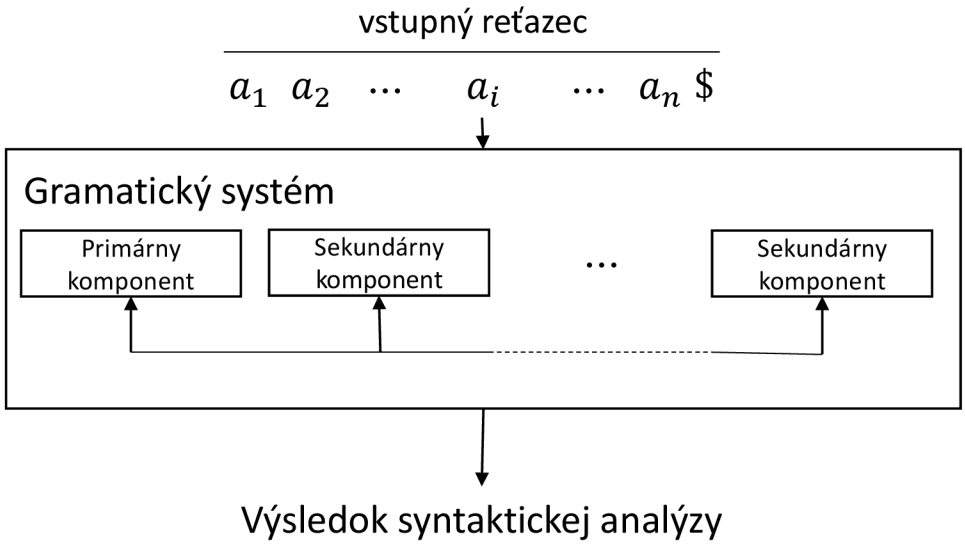
Značným obmedzením precedenčnej syntaktickej analýzy je nemožnosť používať ε pravidlá a pravidlá s rovnakou pravou stranou, ale rozdielnou ľavou stranou. Z tohto dôvodu je nevhodné používať túto metódu inak, ako na overovanie výrazov, čo funguje elegantne pri väčšine programovacích jazykov.

Metódu syntaktyckej analýzy zdola nahor je možné využiť spolu s metódou zhora nadol v syntaktickej analýze implementovanej na základe formálneho modelu gramatického systému.

4.3 Gramatický systém v syntaktickej analýze

Využitie gramatického systému charakterizovaného v Kapitole 3 je v syntaktickej analýze počítačových programov. Komponenty gramatického systému umožňujú kombinovať viacero metód syntaktickej analýzy v jednom formálnom modeli, čo zvyšuje jeho silu a možnosti práce, pretože je možné v priebehu syntaktickej analýzy zmeniť metódu práce. Na proces syntaktickej analýzy sa zvyčajne používajú bezkontextové gramatiky, tak aj v tomto prípade budú komponentami gramatického systému.

Podľa potreby je možné pre každý syntaktický celok vytvoriť samostatný komponent, a tým rozdeliť overovaný jazyk do akýchsi modulov. Každý modul by popisoval inú syntaktickú časť prijímaného jazyka a obsahoval by pravidlá len pre daný celok. Táto možnosť vytvárania modulov pomáha k zlepšeniu čitateľnosti celkovej gramatiky a prípadnú deľbu práce pri väčších a komplikovanejších syntaktických štruktúrach, kedy by bolo obtiažne spravovať jednu veľkú gramatiku s veľkým počtom pravidiel. Gramatický systém v syntaktickej analýze je používaný v jeho centralizovanej variante, kedy je prvý komponent nazývaný primárnym a ostatné sekundárne. Komunikačný krok gramatického systému v centralizovanom variante prebieha vždy s primárnym komponentom, kedy tento komponent buď vedenie odovzdáva, alebo prijíma. Predanie kontextu medzi sekundárnymi komponentmi nie je možné. Tento variant uľahčuje implementáciu a zjednodušuje tým prácu s gramatickým systémom v praxi.



Obr. 4.7: Centralizovaný gramatický systém.

Nasledujú praktickejšie príklady, ktoré budú generovať syntaktické štruktúry z jazyka na konci Kapitoly 5.

Demonštrácia fungovania gramatického systému na praktických príkladoch

Príklady v tejto kapitole dopĺňajú príklady v Sekcii 3.1 o tri, ktoré fungujú na základe metód opísaných v tejto kapitole. V príkladoch bude na generovanie použitý gramatický systém $\Gamma = (G_1, G_2, K, f)$, na základe ktorého bola implementovaná praktická časť tejto práce, definovaný v Prílohe A.

Konvencia 4.3.1. Pre sprehľadnenie označujme neterminálny symbol $STATEMENT\_GLOBAL$ ako STG, $STATEMENT$ ako ST a $STATEMENT\_LIST$ ako STL.

Príklad 4.3.1. Tento príklad sa zaoberá syntaktickou štruktúrou cyklu `do/while`, ktorého syntax je popísaná na konci Sekcie 5.1. Telo cyklu nech obsahuje jeden príkaz `pass` a podmienka jednoduchý výraz $x > 0$. Pred krokom zmeny komponentu je komunikačný symbol podčiarknutý.

$$STG \ {}_g\Rightarrow_{G_1} ST \ STG \ {}_g\Rightarrow_{G_1} do \ \{ \ eol \ ST \ STL \ while \ (\ E \ ) \ eol \ STG \ {}_g\Rightarrow_{G_1} do \ \{ \ eol \ pass \ eol \ STL \ while \ (\ E \ ) \ eol \ STG \ {}_g\Rightarrow_{G_1} do \ \{ \ eol \ pass \ eol \ \} \ eol \ while \ (\ \underline{E} \ ) \ eol \ STG \ {}_c\Rightarrow_{G_2} {}_c\Rightarrow_{G_2} do \ \{ \ eol \ pass \ eol \ \} \ eol \ while \ (\ E \ ) \ eol \ STG \ {}_g\Rightarrow_{G_2} do \ \{ \ eol \ pass \ eol \ \} \ eol \ while \ (\ E \ > \ E \ ) \ eol \ STG \ {}_g\Rightarrow^2_{G_2} do \ \{ \ eol \ pass \ eol \ \} \ eol \ while \ (\ id \ > \ value \ ) \ \underline{eol} \ STG \ {}_c\Rightarrow_{G_1} {}_c\Rightarrow_{G_1} do \ \{ \ eol \ pass \ eol \ \} \ eol \ while \ (\ id \ > \ value \ ) \ eol \ STG \ {}_g\Rightarrow_{G_1} do \ \{ \ eol \ pass \ eol \ \} \ eol \ while \ (\ id \ > \ value \ ) \ eol \ eof$$

Kde terminálny symbol *eol* označuje koniec riadka a *eof* koniec vety (vstupného súboru). Generovaná veta vo forme, ako by bola zapísaná v zdrojovom súbore vyzerá nasledovne:

```
do {
  pass
}
while ( x > 0 )
```

Príklad 4.3.2. Tento príklad demonštruje generovanie výrazu. Výsledok výrazu bude na jednom riadku priradený do premennej.

$$STG \ {}_{g}\Rightarrow_{G_1} ST \ STG \ {}_{g}\Rightarrow_{G_1} TYPE \ id \ = \ ASSIGN \ eol \ STG \ {}_{g}\Rightarrow_{G_1} bool \ id \ = \ ASSIGN \ eol \ STG \ {}_{g}\Rightarrow_{G_1} bool \ id \ = \ \underline{E} \ eol \ STG \ {}_{c}\Rightarrow_{G_2} bool \ id \ = \ E \ eol \ STG \ {}_{g}\Rightarrow_{G_2} bool \ id \ = \ E \ \&\& \ E \ eol \ STG \ {}_{g}\Rightarrow_{G_2} bool \ id \ = \ (\ E \ ) \ \&\& \ E \ eol \ STG \ {}_{g}\Rightarrow_{G_2} bool \ id \ = \ (\ E \ >= \ E \ ) \ \&\& \ E \ eol \ STG \ {}_{g}\Rightarrow^{2}_{G_2} bool \ id \ = \ (\ id \ >= \ value \ ) \ \&\& \ E \ eol \ STG \ {}_{g}\Rightarrow_{G_2} bool \ id \ = \ (\ id \ >= \ value \ ) \ \&\& \ (\ E \ ) \ eol \ STG \ {}_{g}\Rightarrow_{G_2} bool \ id \ = \ (\ id \ >= \ value \ ) \ \&\& \ (\ ! \ E \ ) \ eol \ STG \ {}_{g}\Rightarrow_{G_2} {}_{g}\Rightarrow_{G_2} bool \ id \ = \ (\ id \ >= \ value \ ) \ \&\& \ (\ ! \ id \ ) \ \underline{eol} \ STG \ {}_{c}\Rightarrow_{G_1} bool \ id \ = \ (\ id \ >= \ value \ ) \ \&\& \ (\ ! \ id \ ) \ eol \ STG \ {}_{g}\Rightarrow_{G_1} bool \ id \ = \ (\ id \ >= \ value \ ) \ \&\& \ (\ ! \ id \ ) \ eol \ eof$$

Tento výraz by mohol byť v zdrojovom súbore programu napísaný napríklad ako:

```
bool z = ( x >= 1 ) && ( ! y )
```

Príklad 4.3.3. V syntaktickej štruktúre `enum` je možné použiť dobrovoľné priradenie hodnoty. Tento príklad ukazuje `enum` s dvoma prvkami, kde jednému je priradená hodnota a druhému nie.

$$STG \ {}_{g}\Rightarrow_{G_1} FUNC \ STG \ {}_{g}\Rightarrow_{G_1} enum \ id \ \{ \ id \ ENUM \ STG \ {}_{g}\Rightarrow_{G_1} enum \ id \ \{ \ id \ = \ value \ ENUM\_LIST \ STG \ {}_{g}\Rightarrow_{G_1} enum \ id \ \{ \ id \ = \ value \ , \ id \ ENUM \ STG \ {}_{g}\Rightarrow_{G_1} enum \ id \ \{ \ id \ = \ value \ , \ id \ ENUM\_LIST \ STG \ {}_{g}\Rightarrow_{G_1} enum \ id \ \{ \ id \ = \ value \ , \ id \ \} \ eol \ STG \ {}_{g}\Rightarrow_{G_1} {}_{g}\Rightarrow_{G_1} enum \ id \ \{ \ id \ = \ value \ , \ id \ \} \ eol \ eof$$

V zdrojovom súbore by bol tento príkaz zapísaný napríklad takto:

```
enum x { y = 2 , z }
```

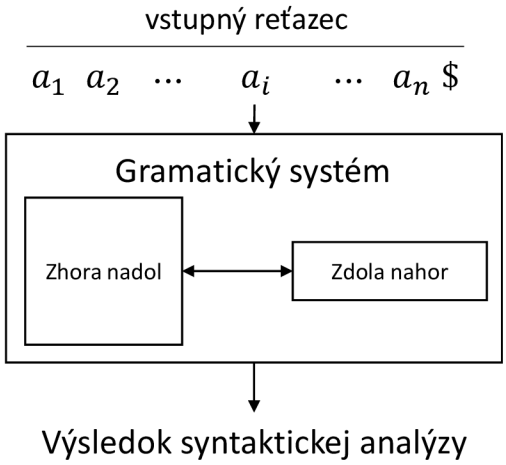
Kapitola 5

Implementácia

Výsledný program je implementovaný ako konzolová aplikácia v jazyku C++, ktorý využíva princípy objektovo orientovaného programovania. Pri implementácii boli využité len štandardné knižnice jazyka C++. Zdrojové súbory sa prekladajú pomocou prekladača `g++` a pomocou nástroja `make`. K programu sú vytvorené aj ukážkové vstupné súbory. V tejto kapitole sa nachádza popis implementácie gramatického systému definovanom v Kapitole 3, popis lexikálneho analyzátora a prijímaného jazyka. Na konci kapitoly je opis vstupu a výstupu programu.

5.1 Moduly programu

Gramatický systém implementovaný ako syntaktický analyzátor sa skladá z dvoch komponentov. Prvý komponent vykonáva syntaktickú analýzu zhora nadol, druhý analýzu zdola nahor.



Obr. 5.1: Schéma implementovaného gramatického systému.

Syntaktická analýza začína od prvého komponentu, ktorému tokeny posiela lexikálny analyzátor a podľa potreby predáva riadenie druhému komponentu. Pri implementácii som sa rozhodol využiť princíp centralizovaného gramatického systému, kde komponent, ktorý vykonáva syntaktickú analýzu zhora nadol, je primárnym. V tomto prípade je výhodné vy-

užiť tento variant, keďže overovaným jazykom je program, ktorého väčšina syntaktických prvkov je overovaná prediktívnym rekurzívnym zostupom, a teda proces overovania syntaxe je výhodné začínať touto metódou. Množinu komunikačných symbolov a mapovaciu funkciu nie je potrebné implementovať. Keďže prvý komponent funguje na základe metódy prediktívneho rekurzívneho zostupu popísanom v Sekcii 4.1, je možné zistiť podľa prepisovacých pravidiel tohoto komponentu, kedy gramatický systém predáva vedenie druhému komponentu. Obdobne druhý komponent predáva vedenie prvému, keď skončí generovanie výrazu úspešne, alebo keď sa v jeho precedenčnej tabuľke nachádza symbol $\times$, ktorý značí chybu. Prvý komponent následne posúdi, či sa jedná naozaj o chybný token, alebo druhý komponent narazil len na neočakávaný token, ktorý avšak prvý očakáva.

Proces fungovania jednotlivých komponentov je popísaný v tejto sekcii, počínajúc lexikálnym analyzátorom, následne syntaktickou analýzou zhora nadol a zdola nahor. Na konci sekcie sa nachádza popis štruktúry a syntaxe prijímajúceho jazyka.

Lexikálny analyzátor

Lexikálny analyzátor bol v tejto práci navrhnutý a implementovaný len na úlohu generovania postupnosti tokenov zo vstupu. Možnosť jeho využitia je teda značne obmedzená, čo sa premieta aj na formát jeho vstupu. Vstupné lexémy musia byť oddelené medzerou a musia spĺňať striktné požiadavky, aby bolo možné uskutočniť mapovanie na príslušný token.

Lexéma	Token	Lexéma	Token
`if`	`IF`	`x, y, z`	`ID`
`switch`	`SWITCH`	`;`	`SEMICOLON`
`1, 2, ...`	`VALUE`	`&&`	`AND`
`{`	`CBL`	`>=`	`GTE`
`)`	`RBR`	`!`	`NOT`

Tabuľka 5.1: Niektoré lexémy z prijímaného jazyka a k nim korešpondujúce tokeny.

V praxi sa obvykle za syntaktickou analýzou nachádza sémantická analýza programu, ktorá by ďalej pracovala so sémantickou stránkou programu a potrebovala by k jednotlivým tokenom viac informácií. Napríklad pri použití typu tokenu `VALUE`, by bola jeho ďalšou zložkou jeho (číselná) hodnota. Viac o implementácii lexikálneho analyzátoru vo svojej plnej forme píšu autori v [1].

Modul syntaktickej analýzy zhora nadol

Modul, ktorý vykonáva syntaktickú analýzu zhora nadol je implementovaný na základe metódy prediktívneho rekurzívneho zostupu popísaného v Sekcii 4.1. Pred samotnou implementáciou bolo potrebné najprv navrhnúť gramatiku jazyka. Z gramatiky sa následne vytvorila LL tabuľka, ktorá bola až na jednu výnimku LL(1) tabuľkou.

Gramatika jazyka umožňuje používanie funkcií namiesto výrazu. Je teda možné pomocou gramatiky syntakticky zapísať priradenie výstupu funkcie do premennej. Keďže lexikálny analyzátor odovzdáva len tokeny, ktoré obsahujú informáciu len o type, tak nie je možné zistiť, či token označujúci identifikátor označuje funkciu, alebo inú premennú, ktorá môže byť použitá vo výraze. Táto komplikovaná situácia bola vyriešená implementovaním funkcie `peek_token` v lexikálnom analyzátore, ktorá umožňuje nahliadnutie na nasledujúci token, ktorý avšak nie je spracovaný, ale ostáva stále na vstupe. Lexikálny analyzátor takto

dokáže predať kópiu tohto tokenu ľubovoľnému komponentu, ktorý si o to zažiada. Túto možnosť využíva len komponent syntaktickej analýzy zhora nadol a to v prípade rozoznania či ide o výraz, alebo o volanie funkcie a rozoznania holého výrazu na riadku od priradenia hodnoty do premennej. Komponent potom len overuje, či ide o token `(`, alebo `=` a podľa toho vie, ako má pokračovať.

Modul syntaktickej analýzy zdola nahor

Komponent, ktorý má na starosť metódu precedenčnej syntaktickej analýzy zdola nahor pracuje na základe Algoritmu 4.2.1. Na správne fungovanie algoritmu bolo potrebné ručne vytvoriť tabuľku precedencie podľa postupu opísaného v Sekcii 4.2. Precedenčný syntaktický analyzátor podporuje značnú časť operátorov používaných v bežných programovacích jazykoch. Podporované sú aritmetické, relačné a logické operátory vrátane unárneho operátora negácie. Tabuľka precedencie je z dôvôdu jej veľkosti uvedená v Prílohe A.

Dôležitou pomocnou časťou tohto modulu je jeho zásobník, ktorý musel byť implementovaný ako nadstavba nad bežnou implementáciou zásobníka, ktorý obsahuje štandardná knižnica jazyka C++. Hlavným dôvodom na obohatenie implementácie bežného zásobníka je potreba nálezu handle, ktorá sa využíva pri operácii redukcie. Zásobník teda počínajúc od svojho vrcholu vráti vektor symbolov až po znak $<$, ktorý značí začiatok reťazca pripraveného na redukciu. Tento vektor následne preberie modul precedenčnej syntaktickej analýzy, ktorý skontroluje, či existuje pravidlo s pravou stranou rovnou tomuto vektoru. Zásobník tiež obsahuje mapovanie tokenov na symboly, ktoré si môže zásobník ukladať. Jedným z dôvodov zavedenia tohto mapovania je uľahčenie práce so zásobníkom. Pri výskyte tokenu `ID`, alebo `VALUE` si zásobník tento token namapuje na svoj vlastný symbol `S_I`, ktorý v tabuľke precedencie označuje identifikátor. Vďaka tomuto mapovaniu nastáva sprehľadnenie a zjednodušenie kódu precedenčného syntaktického analyzátora. Ďalším špeciálnym symbolom, ktorý sa môže vyskytnúť na zásobníku je symbol `S_E`, ktorý označuje neterminálny symbol E používaný v pravidlách gramatiky syntaktického analyzátora, symbol `S_SHIFT`, ktorý reprezentuje špeciálny znak $<$ používaný pri operácii $SHIFT(<)$ a symbol `S_BOTTOM`, ktorý označuje dno zásobníka (špeciálny znak \$ v Algoritme 4.2.1). Zvyšok symbolov, ktoré je možné vložiť do zásobníka sú operátory a zátvorky, s ktorými sa precedenčný syntaktický analyzátor môže bežne dostať do styku.

Prijímaný jazyk

Podporované syntaktické štruktúry sú z jazyka IFJ19[1], ktoré boli syntakticky pozmenené na štýl jazyka C/C++ a obohatené o rozšírenia ako je napríklad cyklus `do/while`, alebo štruktúru `switch`. Hlavnou zmenou bola zmena ohraničenia tela cyklov a iných štruktúr zo zmeny odsadenia na použitie zložených zátvoriek. Ďalej boli odstránené niektoré štruktúry, ktoré by neboli z hľadiska syntaktickej analýzy zaujímavé. Z jazyka IFJ19 ostalo oddeľovanie jednotlivých príkazov znakom konca riadka. Nasleduje bližšie popísanie zavedených rozšírení:

- Cyklus `do/while`:
  Telo cyklu `do/while` je ohraničené zloženými zátvorkami, medzi ktorými sa musí nachádzať aspoň jeden príkaz. Za telom sa nachádza kľúčové slovo `while` nasledované výrazom v zátvorkách. Syntax tejto štruktúry je nasledujúca:

[1] špecifikácia jazyka dostupná na https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Fprojects%2FHistory%2Fifj2019.pdf&cid=13305

```
do {
  <sekvencia príkazov>
}
while ( <výraz> )
```

- Zreťazený podmienený príkaz:
  Podmienený príkaz charakterizovaný kľúčovým slovom `if` bol rozšírený o zreťazenie podmienok pomocou dvoch kľúčových slov `else` a `if` na jednom riadku. Toto zreťazenie sa môže vyskytovať v ľubovoľnom počte. Pred posledným blokom tejto syntaktickej štruktúry sa musí nachádzať len kľúčové slovo `else`. Syntax štruktúry s jedným výskytom `else if` je nasledujúca:

```
if ( <výraz> ) {
  <sekvencia príkazov>
}
else if ( <výraz> ) {
  <sekvencia príkazov>
}
else {
  <sekvencia príkazov>
}
```

- Cyklus `for`:
  Ďalším rozšírením jazyka je cyklus `for` inšpirovaný syntaxou z jazyka C. Za kľúčovým slovom `for` sa nachádzajú v zátvorke tri bloky oddelené bodkočiarkou, z ktorých prvý blok musí byť priradenie výrazu alebo výstupu z funkcie do premennej a ostatné dva sú výrazmi. Syntax štruktúry je nasledujúca:

```
for ( <typ> <id> = <priradenie> ; <výraz> ; <výraz> ) {
  <sekvencia príkazov>
}
```

  kde `<typ>` a `<id>` vyjadrujú použitie lexém, ktoré reprezentujú tokeny pre dátový typ a identifikátor.

- Syntaktická štruktúra `switch`:
  Použitie štruktúry `switch` je podobné so syntaxou v jazyku C. Za kľúčovým slovom `switch` je v zátvorke výraz nasledujúci blokom, ktorý obsahuje ľubovoľný počet blokov, ktoré začínajú kľúčovým slovom `case`. Ako posledným v tele štruktúry `switch` sa musí nachádzať blok s označením `default`. Syntax štruktúry s jedným výskytom `case` je nasledujúca:

```
switch ( <výraz> ) {
  case <hodnota> {
    <sekvencia príkazov>
  }
  default {
    <sekvencia príkazov>
  }
```

kde `<hodnota>` vyjadruje použitie lexémy, ktorá reprezentuje token `VALUE`.

- Syntaktická štruktúra `enum`:
  Posledným rozšírením k jazyku IFJ19 je štruktúra `enum`, ktorej syntax je taktiež prebratá z jazyka C. Celý príkaz sa vyskytuje na jednom riadku, je označený kľúčovým slovom `enum` a identifikátorom. Ďalej sa nachádza telo, v ktorom je možné vymenovať identifikátory oddelené čiarkou, pri ktorých sa môže dobrovoľne nachádzať priradenie konkrétnej hodnoty. Syntax štruktúry s dvoma výskytmi identifikátorov je nasledujúca:

  ```
  enum <id> { <id> , <id> = <hodnota> }
  ```

Zotavenie z chýb

Komponent syntaktickej analýzy zhora nadol disponuje schopnosťou pokračovať v analýze po narazení na chybu. Spustenie analyzátora v tzv. zotavovacom móde je potrebné spustiť špeciálnou metódou `parseInRecoveryMode`, ktorá prejde celý vstup a vypíše každú nájdenú chybu. Nie je zaručené, že chybová hláška bude popisovať naozajstnú chybu, avšak okolo popísaného tokenu sa chyba pravdepodobne vyskytuje.

Zotavovanie prebieha na princípe techniky preskakovania vstupných tokenov popísanej na konci Sekcie 4.1. Z implementačného pohľadu bola množina T_Γ synchronizačných tokenov zvolená ako zjednotenie množiny $First(STATEMENT)$ a tokenu `ENDOFFILE`, ktorý značí koniec vstupu. Množina T_Γ teda obsahuje nasledujúce tokeny:

$$T_\Gamma = \left\{ \begin{array}{llll} \texttt{ID}, & \texttt{FOR}, & \texttt{WHILE}, & \texttt{SWITCH}, \quad \texttt{DO}, \\ \texttt{PASS}, & \texttt{BREAK} & \texttt{CONTINUE}, & \texttt{RETURN} \\ \texttt{INT}, & \texttt{DOUBLE}, & \texttt{ENDOFFILE}, & \texttt{BOOL} \end{array} \right\}. \tag{5.1}$$

Syntaktický analyzátor obnoví svoju činnosť až po narazení na jeden z týchto tokenov a spustí svoju analýzu od neterminálu $STATEMENT\_GLOBAL$. Tento začiatok som volil z dôvodu plného obnovenia rekurzívnej metódy a ukončenia analýzy v prípade narazenia na koniec vstupu. Pri zvolení tohto neterminálu nastáva viacero problémov. Prvým je obnovenie analýzy vo vnorenom bloku. Každý blok je skončený tokenom `}`, ktorý by nemusel byť na chybnom mieste, ale v tomto prípade syntaktický analyzátor nevie na ktorej úrovni zanorenia sa nachádza. Z tohoto dôvodu je tento token preskočený, ak sa v zotavovacom móde objaví na vstupe v procedúre, ktorá reprezentuje neterminál $STATEMENT\_GLOBAL$. Syntaktický analyzátor sa tak môže vynoriť a pokračovať v analýze.

Ďalším problémom je výskyt kľúčového slova `while` v dvoch syntaktických štruktúrach. Ak sa po chybe obnoví analýza v tele cykla `do/while` bude kľúčové slovo `while` považované

ako začiatok rovnomenného cykla a syntaktický analyzátor vyhlási chybu na konci riadka, ktorá avšak chybou byť nemusí.

Tretím problémom je možný výskyt definície funkcie vo vnorenom bloku, čo jazyk nedovoľuje, ale syntaktický analyzátor v zotavovacom móde túto chybu neodhalí. Tento typ syntaktickej chyby avšak považujem za veľmi zriedkavý.

V každom z týchto prípadov sa analyzátor nachádza v zotavovacom móde, narazil už na chybu a prehlásil vstupný súbor za syntakticky nesprávny. Po opravení predošlej chyby (chýb) a spustení analyzátora v normálnom móde sa tieto chyby určite odhalia. Výhodou zotavovenia z chýb je možné odhalenie všetkých syntaktických prehreškov, ktoré sa vo vstupnom súbore nachádzajú. Táto výhoda značne prevyšuje možný výskyt spomenutých chýb v jej implementácii. Ďalšia sekcia obsahuje popis tvaru chybových hlášok a vstupu a výstupu celého programu.

5.2 Vstup a výstup

Táto sekcia popisuje užívateľské rozhranie programu. Opisuje formát vstupného súboru, výstup syntaktického analyzátora a formát spustenia programu.

Vstup programu

Načítavanie zo vstupu zabezpečuje kompletne lexikálny analyzátor, ktorý lexémy oddelené medzerou mapuje na tokeny. Tento vstup berie zo štandardného vstupu, kde každý riadok ukončí tokenom `EOL` a celý vstup tokenom `EOF`. Vstupný súbor je odporúčané presmerovať pomocou znaku `<` v konzolovom jazyku Bash. Z dôvodu obmedzenia funkčnosti lexikálneho analyzátoru, je povolené mapovanie lexém `x`, `y` a `z` na token `ID` a jednociferné čísla spolu s lexémami `true` a `false` na token `VALUE`. V gramatike jazyka tieto tokeny označujú terminálne symboly *id* a *value*. Príklad vstupného súboru môže vyzerať nasledovne:

```
enum x { x = 1 , y , z = 5 }
for ( int x = 0 ; x < 5 ; x + 1 ) {
  if ( ( x / 2 ) == 0 ) {
    print(x)
  }
  else {
    continue
  }
}
```

Jednotlivé syntaktické konštrukcie nemusia zo sémantickej stránky dávať zmysel, ale cieľom práce je overovanie syntaxe zdrojového programu. Projekt obsahuje vzorové vstupné súbory a skript, ktorý automaticky spustí testy v priečinku `examples`.

Výstup programu

Syntaktický analyzátor po úspešnom overení syntaxe vstupného súboru skončí s návratovou hodnotou 0 a na výstup vypíše hlášku `Syntax ok`. V opačnom prípade vráti hodnotu 2 ak nastala chyba v analyzátore zhora nadol, 3 ak nastala chyba v analyzátore zdola nahor a vypíše chybovú hlášku, v ktorej je chyba bližšie špecifikovaná. Lexikálny analyzátor rovnako

vracia chybovú hlášku, ak narazí na neznámu lexému, ktorú nevie namapovať. Príklady týchto hlášok sú:

```
LEXICAL ERROR 1: Unknown token 'fro' on line 18!
SYNTAX ERROR 2: Unexpected token '=' on line 20! Expected token is '('.
SYNTAX ERROR 3: No rule for an expression found on line 18!
```

Pri výskyte viacerých chýb sa vracia návratová hodnota určená podľa prvej chyby. Na výstupe sa potom objaví hláška pre každú chybu. Výsledky analýzy a chybové hlášky sa vypisujú na štandardný chybový výstup. Pri použití programu s prepínačom `-r` sú na štandardný výstup vypísané prepisovacie pravidlá komponentov v poradí, v akom boli aplikované.

Formát spustenia

Program je implementovaný vo forme príkazovej riadky v operačnom systéme Linux. Odporúčaný formát spustenia programu je v jazyku Bash použitím presmerovania:

```
./main < INPUT
```

kde `INPUT` je cesta k vstupnému súboru. Program disponuje schopnosťou vypísať pravidlá, ktoré boli použité počas procesu syntaktickej analýzy. V tomto prípade je odporúčaný formát spustenia v jazyku Bash nasledovný:

```
./main -r < INPUT > OUTPUT
```

kde `INPUT` a `OUTPUT` sú cesty k súborom so vstupom a výstupom programu.

Kapitola 6

Záver

Cieľom tejto práce bolo vytvorenie formálneho modelu nového gramatického sytému, ktorý dokáže kombinovať niekoľko metód syntaktickej analýzy a následne ho implementovať. Tento cieľ sa podarilo naplniť, a hneď po uvedení základných pojmov, je model gramatického systému formalizovaný. Inšpiráciou k tvorbe gramatického systému bolo komunikovanie komponentov prevzaté z kooperačne distribuovaných (CD) gramatických systémov. Formálny model doplňujú príklady generovania viet bezkontextového a kontextového jazyka pomocou gramatického systému.

Ďalej sa rozoberá proces syntaktickej analýzy, jej metódy zhora nadol a zdola nahor a ich využitie v syntaktických analyzátoroch. Hovorí sa aj o aplikácii navrhnutého gramatického systému v syntaktickej analýze, ktorý narozdiel od klasických metód dokáže kombinovať viaceré typy metód a v priebehu analýzy medzi nimi prepínať. Rozoberá sa generovanie a analýza syntaktických štruktúr pomocou gramatického systému z jazyka, ktorý bol navrhnutý v praktickej časti práce.

Následne som gramatický systém navrhol a implementoval. Po konzultácii s vedúcim sme od pôvodného návrhu overovania nebezkontextových štruktúr paralelne v skorých štádiách upustili, a ďalej sme sa sústredili na overovanie bezkontextových štruktúr sekvenčne. Výsledný gramatický systém obsahuje dva komponenty, jeden funguje na základe metódy zhora nadol a druhý zdola nahor. Implementačne sa jedná o variant centralizovaného gramatického systému, kde komponent vykonávajúci analýzu zhora nadol je primárny. Výslednú aplikáciu som testoval pomocou vytvoreného testovacieho skriptu a vstupných súborov, ktoré obsahovali syntaktické štruktúry z overovaného jazyka a ich kombinácie.

Nad rámec zadania bolo implementované zotavenie syntaktického analyzátora zhora nadol pri nájdení chyby. S využitím techniky preskakovania tokenov až po nájdenie synchronizačného tokenu je syntaktický analyzátor schopný v móde zotavovania odhaliť viacero chýb v priebehu analýzy.

Práca sa snaží poskytnúť základy v oblasti gramatických systémov, syntaktickej analýzy a prepojenia týchto dvoch odvetví. V práci by ešte niekto mohol pokračovať rozšírením terajšieho, prípadne zadefinovaním nového modelu gramatického systému, ktorý bude fungovať paralelne.

Literatúra

[1] Aho, A., Lam, M., Sethi, R. a Ullman, J. *Compilers : principles, techniques, and tools*. Second edition. Harlow: Pearson, 2014. ISBN 978-1-292-02434-9.

[2] Chomsky, N. On certain formal properties of grammars. *Information and control*. 1. vyd. Elsevier. 1959, zv. 2, č. 2, s. 137–167.

[3] Meduna, A. *Automata and Languages: Theory and Applications*. 1. vyd. Springer Verlag, 2005. 892 s. ISBN 1-85233-074-0. Dostupné z: https://www.fit.vut.cz/research/publication/6177.

[4] 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.

[5] Meduna, A. a Lukáš, R. *Syntaktická analýza shora dolů* [online]. FIT VUT v Brně, 2017 [cit. 2022-04-06]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj07-cz.pdf.

[6] Meduna, A. a Lukáš, R. *Syntaktická analýza zdola nahoru* [online]. FIT VUT v Brně, 2017 [cit. 2022-04-09]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj08-cz.pdf.

[7] Češka, M., Vojnar, T., Smrčka, A. a Rogalewicz, A. *Teoretická informatika: Studijní text* [online]. FIT VUT v Brně, august 2020. Dostupné z: http://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf.

Príloha A

Gramatický systém použitý na syntaktickú analýzu

Na syntaktickú analýzu bol použitý centralizovaný gramatický systém $\Gamma = (G_1, G_2, K, f)$, kde pre prehľadnosť je uvedená len množina prepisovacích pravidiel, množina komunikačných symbolov a mapovacia funkcia f. Zvyšné množiny je možné odvodiť z množín prepisovacích pravidiel, počiatočný nonterminál je na ľavej strane prvého pravidla.

Prepisovacie pravidlá komponentu G_1

Tento komponent vykonáva syntaktickú analýzy zhora nadol metódou prediktívneho rekurzívneho zostupu.

```
STATEMENT_GLOBAL -> FUNC STATEMENT_GLOBAL
STATEMENT_GLOBAL -> STATEMENT STATEMENT_GLOBAL
STATEMENT_GLOBAL -> eof
STATEMENT_LIST   -> STATEMENT STATEMENT_LIST
STATEMENT_LIST   -> } eol

FUNC          -> def TYPE id ( PARAM_LIST { eol STATEMENT STATEMENT_LIST
FUNC          -> enum id { id ENUM
ENUM          -> ENUM_LIST
ENUM          -> = value ENUM_LIST
ENUM_LIST     -> } eol
ENUM_LIST     -> , id ENUM

STATEMENT     -> TYPE id = ASSIGN eol
STATEMENT     -> id = ASSIGN eol
STATEMENT     -> ASSIGN eol
STATEMENT     -> if ( E ) {  eol STATEMENT STATEMENT_LIST else ELSIF
ELSIF         -> if ( E ) { eol STATEMENT STATEMENT_LIST else ELSIF
ELSIF         -> { eol STATEMENT STATEMENT_LIST
STATEMENT     -> for ( TYPE id = ASSIGN ; E ; E ) { eol STATEMENT STATEMENT_LIST
STATEMENT     -> while ( E ) { eol STATEMENT STATEMENT_LIST
STATEMENT     -> do { eol STATEMENT STATEMENT_LIST while ( E ) eol
STATEMENT     -> switch ( E ) { eol CASE
```

```
CASE          -> case value { eol STATEMENT STATEMENT_LIST CASE
CASE          -> default { eol STATEMENT STATEMENT_LIST } eol
CASE          -> } eol
STATEMENT     -> pass eol
STATEMENT     -> break eol
STATEMENT     -> continue eol
STATEMENT     -> return RETURN_E

RETURN_E      -> eol
RETURN_E      -> E eol

PARAM_LIST    -> TYPE id PARAM
PARAM_LIST    -> )
PARAM         -> , TYPE id PARAM
PARAM         -> )

ASSIGN        -> id ( ARG_LIST
ASSIGN        -> E

ARG_LIST      -> VALUE ARG
ARG_LIST      -> )
ARG           -> , VALUE ARG
ARG           -> )
TYPE          -> int
TYPE          -> double
TYPE          -> bool

VALUE         -> value
VALUE         -> id
```

Prepisovacie pravidlá komponentu G_2

Tento komponent vykonáva syntaktickú analýzy zdola nahor metódou precedenčnej syntaktickej analýzy.

```
E   -> i
E   -> ! E
E   -> ( E )
E   -> E ^ E
E   -> E * E
E   -> E / E
E   -> E + E
E   -> E - E
E   -> E < E
E   -> E <= E
E   -> E > E
E   -> E >= E
E   -> E == E
E   -> E != E
```

```
E   -> E && E
E   -> E || E
```

	^	*	/	+	−	*i*	(	)	>	>=	<	<=	==	! =	&&	||	!	$
^	<	>	>	>	>	<	<	>	>	>	>	>	>	>	>	>	<	>
*	<	>	>	>	>	<	<	>	>	>	>	>	>	>	>	>	<	>
/	<	>	>	>	>	<	<	>	>	>	>	>	>	>	>	>	<	>
+	<	<	<	>	>	<	<	>	>	>	>	>	>	>	>	>	<	>
−	<	<	<	>	>	<	<	>	>	>	>	>	>	>	>	>	<	>
i	>	>	>	>	>	×	×	>	>	>	>	>	>	>	>	>	>	>
(	<	<	<	<	<	<	<	=	<	<	<	<	<	<	<	<	<	×
)	>	>	>	>	>	×	×	>	>	>	>	>	>	>	>	>	>	>
>	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>
>=	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>
<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>
<=	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>
==	<	<	<	<	<	<	<	>	<	<	<	<	>	>	>	>	<	>
! =	<	<	<	<	<	<	<	>	<	<	<	<	>	>	>	>	<	>
&&	<	<	<	<	<	<	<	>	<	<	<	<	<	<	>	>	<	>
||	<	<	<	<	<	<	<	>	<	<	<	<	<	<	>	>	<	>
!	>	>	>	>	>	<	<	>	>	>	>	>	>	>	>	>	<	>
$	>	>	>	>	>	>	>	×	>	>	>	>	>	>	>	>	>	✓

Tabuľka A.1: Precedenčná tabuľka používaná druhým komponentom.

Množina K a funkcia f

Množina komunikačných symbolov je definovaná nasledovne:

$$K = \{eol, E\}.$$

Mapovacia funkcia f je definovaná nasledovne:

K	$\mathbf{G_i}$
eol	G_1
E	G_2

Príloha B

Obsah priloženého pamäťového média

Pamäťové médium obsahuje zdrojové súbory v jazyku C++, manuál k spusteniu aplikácie a zdrojové súbory práce pre nástroj LaTeX. Štruktúra súborov je nasledujúca:

- `thesis` – priečinok obsahujúci zdrojové súbory práce a PDF práce,
- `src` – priečinok obsahujúci zdrojové súbory programu:
  - `common` – priečinok obsahujúci implementáciu zásobníku,
  - `examples` – priečinok obsahujúci ukážkové súbory a testovací skript,
  - `parsers` – priečinok obsahujúci jednotlivé komponenty gr. systému,
  - `Lexer.cpp/.h` – zdrojové súbory implementujúce lexikálny analyzátor,
  - `Main.cpp` – hlavná trieda programu z ktorej sa spúšťa analýza,
  - `makefile` – súbor na preklad programu,
  - `readme.txt` – súbor obsahujúci popis prekladu a spustenia aplikácie.