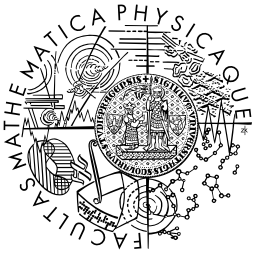


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Bc. Tomáš Dzurenko

Generátor přívětivých analyzátorů

Středisko informatické sítě a laboratoří

Vedoucí diplomové práce:	RNDr. Michal Žemlička, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové systémy

Praha 2016

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 27.7.2016 Bc. Tomáš Dzurenko

Názov práce: Generátor přívětivých analyzátorů

Autor: Bc. Tomáš Dzurenko

Katedra: Středisko informatické sítě a laboratoří

Vedúci diplomovej práce: RNDr. Michal Žemlička, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Táto práca predstavuje generátor umožňujúci z popisu k-prívetivej gramatiky vytvoriť zdrojový kód implementujúci analyzátor alebo translátor pre jazyk generovaný touto gramatikou. Oproti klasickým *LL(k)* gramatikám umožňujú k-prívetivé gramatiky použiť v pravidlách priamu ľavú rekurziu. To umožňuje pohodlnejšiu a prehľadnejšiu formuláciu pravidiel vstupnej gramatiky.

Kľúčové slová: analyzátor, translátor, generátor, C++, prívetivá gramatika

Title: Kind parser generator

Author: Bc. Tomáš Dzurenko

Department: Network and Labs Management Center

Supervisor: RNDr. Michal Žemlička, Ph.D., Department of Software and Computer Science Education

Abstract: This thesis introduces a generator which takes as its input a definition k-kind grammar and creates source code of analyzer or translator for the language generated by this kind grammar. Opposed to traditional *LL(k)* grammars, k-kind grammars allow usage of direct left recursion in its rules. This allows for more comfortable and clearer formulation of input grammar rules.

Keywords: parser, translator, generator, C++, kind grammar

Ďakujem vedúcemu diplomovej práce RNDr. Michalovi Žemličkovi, Ph.D. za poskytnutie literatúry, odborné vedenie, cenné rady a pripomienky pri vypracovaní mojej diplomovej práce.

Obsah

Kapitola 1

Úvod

Cieľom tejto práce je implementácia generátora vytvárajúceho zdrojový kód analyzátorov a translátorov na základe definície prívetivej gramatiky. Hlavný dôraz je pritom kladený na nasledujúce vlastnosti vytváraných analyzátorov a translátorov:

1. *Jednoduchosť a prehľadnosť zdrojového kódu*

   Vygenerovaný zdrojový kód by mal byť bežnému programátorovi bez problémov zrozumiteľný a mal by sa v ňom dokázať čo najjednoduchšie zorientovať, aby mohol efektívne prevádzať prípadné úpravy. Ideálne by mal zdrojový kód pripomínať ručne písaný program a mal by byť vhodne rozdelený na logické celky (súbory, triedy, metódy). K prehľadnosti vygenerovaného zdrojového kódu by ďalej mala prispiesť aj dokumentácia, ktorá bude vygenerovaná priamo v zdrojovom kóde.

2. *Užívateľská príjemnosť*

   Z pohľadu užívateľa by mal byť vygenerovaný analyzátor (translátor) čo najpríjemnejší na použitie. To znamená, že užívateľské rozhranie by malo byť jednoduché, intuitívne a malo by vyžadovať minimálne množstvo nutných nastavení potrebných k použitiu analyzátora.

3. *Minimálna funkčnosť*

   Jadro vygenerovaného zdrojového kódu by malo poskytovať len minimálnu funkčnosť potrebnú k zaisteniu procesu parsovania. Všetka ostatná funkcionalita by mala byť doplniteľná užívateľom (pomocou sémantických akcií). Zároveň by mal byť zabezpečený mechanizmus ako túto funkcionalitu jednoducho zakomponovať do výsledného zdrojového kódu počas procesu generovania analyzátora (translátora).

4. *Uniformita*

   Štruktúra jednotlivých vygenerovaných analyzátorov by mala byť nápadne podobná. Ideálne by mali byť spoločné časti zdrojového kódu oddelené od tých špecifických a uložené v samostatných súboroch. Táto vlastnosť by ďalej uľahčila prípadne úpravy nad celou množinou vygenerovaných analyzátorov.

5. *Bezproblémová spolupráca s ďalšími vygenerovanými analyzátormi*

   Jednotlivé vygenerované analyzátory by mali byť schopné spolupracovať. Spoluprácou sa myslí, že každý analyzátor by mal byť bez problémov schopný použiť inú inštanciu analyzátora a využívať výsledok jeho práce.

1.1 Zadanie

Táto práca nadväzuje[1] na implementáciu prívetivého konštruktora *KindCons* [8], ktorý je schopný generovať analyzátory a translátory v jazyku *C*. V tejto práci sa pokúsime rozšíriť a vylepšiť výsledný produkt generátora z hľadiska funkcionality a užívateľskej príjemnosti a tým sa čo najviac priblížiť reálne použiteľným analyzátorom a translátorom.

Vytýčené cieľe sumarizuje nasledovné zadanie tejto práce:

Vytvořte generátor syntaktických analyzátorů, případně i translátorů vytvářející k zadanému popisu přívětivé gramatiky kód v C++. Nástroj by měl podporovat gramatiky s výhledem nejen 1, ale i delším. I v těchto případech by mělo být zachováno pravidlo, že generovaný kód je snadno čitelný i modifikovatelný.

Zvažte podporu importu a exportu popisu jazyka z a do podoby používané jinými generátory syntaktických analyzátorů.

1.2 Štruktúra práce

Text tejto práce je rozdelený do siedmich kapitol. Nasledujúca kapitola poskytuje stručný prehľad o analyzátoroch a ich generátoroch a uvádza príbuzné práce zaoberajúce sa generovaním analyzátorov.

V prvej časti tretej kapitoly sú uvedené niektoré základné pojmy z teórie jazykov a prekladačov. Druhá časť tretej kapitoly potom zhrňuje teoretické poznatky použité pri návrhu a implementácii prívetivých analyzátorov a translátorov.

Štvrtá kapitola sa podrobnejšie zaoberá výpočtom výhľadových množín a výhľadov, návrhom štruktúr, ktoré tieto výhľady využívajú a generovaním zdrojového kódu z týchto štruktúr.

Piata kapitola približuje samotnú aplikáciu CppKind, ktorá predstavuje implementáciu generátora prívetivých analyzátorov a translátorov.

Šiesta kapitola podrobne predstavuje štruktúru vygenerovaných prívetivých analyzátorov a translátorov z pohľadu programátora a zároveň poskytuje ukážku použitia vygenerového translátora.

Záverečná siedma kapitola obsahuje zhrnutie dosiahnutých výsledkov a možnosti ďalšieho rozvoja.

[1] Aj keď bol zdrojový kód programu *KindCons* autorovi tejto práce prístupný, nebola žiadna časť pôvodného zdrojového kódu priamo prevzatá a jedná sa o implementáciu vytvorenú úplne od základu. V niektorých špecifických častiach však implementácia pripomína pôvodnú implementáciu, nakoľko sú riešené podobné problémy.

Kapitola 2

Generátory analyzátorov a príbuzné práce

Generovanie lexikálnych a syntaktických analyzátorov sa robí už dlhšiu dobu (jedny z najstarších sú *Yacc* [18] a *Lex* [19]). Existuje mnoho rôznych nástrojov umožňujúcich vygenerovať k zadanej gramatike kód analyzátora, či priamo podľa zadania gramatiky analyzovať vstup. Tu sa zameriame na vybrané nástroje, ktoré generujú analyzátory pracujúce rekurzívnym zostupom, s výhľadom väčším ako 1 alebo majú prostriedky na riešenie konfliktov výhľadu dĺžky 1.

V tejto kapitole najskôr predstavíme niektoré zaužívané pojmy spojené s analyzátormi a zľahka popíšeme ich štruktúru a funkčnosť (s týmito a mnohými ďalšími pojmami je možné sa bližšie zoznámiť v literatúre venujúcej sa konštrukcii analyzátorov a kompilátorov, napríklad v [2]). Následne spomenieme niektoré dostupné nástroje zaoberajúce sa generovaním analyzátorov.

Generátor analyzátorov je nástroj, ktorý na základe popisu štruktúry vstupných dát vytvorí zdrojové kódy analyzátora, ktorý je schopný tieto dáta spracovávať. Analyzátor sa zvyčajne skladá z dvoch častí.

Prvou časťou je *lexikálny analyzátor* [2, str. 109], ktorého úlohou je vo vstupných dátach rozpoznávať prípustné reťazce nazývané *lexémy* [2, str. 111]. Tieto reťazce (prípadne množiny reťazcov) majú presne určený *vzor* [2, str. 111], ktorý môže byť definovaný napríklad pomocou regulárnych výrazov. Z nájdených lexémov vytvára lexikálny analyzátor objekty nazývané *tokeny* [2, str. 111], ktoré identifikujú typ rozpoznaného lexému. Vytvorené tokeny sú následne predávané druhej časti analyzátora, ktorou je *syntaktický analyzátor* [2, str. 192].

Syntaktický analyzátor slúži ku kontrole syntaktickej štruktúry vstupných dát, ktorá je definovaná pomocou formálnej gramatiky. Jeho vstupom sú tokeny vytvorené lexikálnym analyzátorom. Názvy terminálov gramatiky odkazujú na identifikátory uložené v týchto tokenoch. Súbor pravidiel formálnej gramatiky potom špecifikuje syntaktickú podobu vstupných dát. Syntaktická kontrola je vykonávaná aplikáciou týchto pravidiel na tokeny. S jednotlivými pravidlami gramatiky môžu byť asociované časti výkonného kódu, ktorý je prevádzaný pri použití príslušného pravidla. Tieto časti kódu nazývame *sémantické akcie*, pretože definujú sémanticku stránku syntaktickej štruktúry analyzovanej syntaktickým analyzátorom.

Oproti naprogramovanému analyzátoru má vygenerovaný analyzátor niekoľko výhod. Vygenerovaný analyzátor je napríklad ľahko *modifikovateľný*, pretože pri

zmene štruktúry vstupných dát je nutné zmeniť len popis tejto štruktúry namiesto prepisovania samotného analyzátora. Oproti programovým konštruktom predstavuje špecifikácia vstupných dát *zrozumiteľnejšiu* formu pre pochopenie štrukúry vstupných dát. Vygenerovanie analyzátora pomocou overeného generátora tiež predstavuje *nižšie riziko výskytu chýb*, ako pri jeho ručnom programovaní.

Prehľadnosť a čitateľnosť je dôležitá vlastnosť zdrojového kódu. Ak je kód vygenerovaného analyzátora prehľadný a zrozumiteľný, je ďaleko jednoduchšie do neho vložiť dodatočnú funkcionalitu – napríklad ďalšie sémantické akcie alebo riešenie chybových stavov. Zároveň to zjednodušuje údržbu vygenerovaného analyzátora v prípade, že z nejakého dôvodu už nie je možné alebo vhodné analyzátor pregenerovať. Okrem toho je v zrozumiteľnom kóde jednoduchšie overiť, že naozaj robí to, čo sa od neho očakáva, čím sa zvyšuje jeho dôveryhodnosť.

V dnešnej dobe existuje celá rada generátorov analyzátorov, ktoré generujú analyzátory pre rôzne programovacie jazyky a platformy. Tieto generátory využívajú rôzne stratégie analýzy vstupného textu. Väčšina dnešných generátorov generuje analyzátory založené buď na *LL(1)* [2, str. 222] alebo *LR(1)* [2, str. 241] gramatikách. Výhody práce s gramatikami používajúcimi výhľad dlhší ako 1 rozoberá napríklad práca [14].

LL(k) analyzátory

LL(k) analyzátory analyzujú vstup zľava doprava, pričom produkujú najľavejšiu deriváciu (viď definície 3.4 a 3.5). Symbol k označuje dĺžku výhľadu potrebnú k výberu správneho pravidla gramatiky. Syntaktická analýza využíva zásobník, do ktorého je na začiatku analýzy vložený počiatočný neterminál. V ďalšej fáze sa pokračuje na základe symbolu uloženého na vrchole zásobníka.

V prípade, že sa na vrchole zásobníka nachádza terminál, je tento terminál porovnaný s aktuálnym tokenom a v prípade zhody sa token aj terminál odstránia a pokračuje sa nasledujúcim symbolom na zásobníku. Ak sa terminál nezhoduje s aktuálnym tokenom, je oznámená chyba.

V prípade, že sa na vrchole zásobníka nachádza neterminál, nahradí sa pravou stranou pravidla, ktoré má tento neterminál na svojej ľavej strane. V prípade, že je takých pravidiel viac, musí byť správne pravidlo určené pomocou aktuálneho výhľadu. V prípade, že na základe výhľadu nejde určiť správne pravidlo, je znova oznámená chyba.

Analýza končí v momente, kedy sa zásobník úplne vyprázdni a na vstupe už nie je žiadny token (v prípade, že musí byť spracovaný celý vstup).

Vygenerovaný *LL(k) analyzátor* je prehľadný a jednoducho modifikovateľný. Analyzátor môže byť implementovaný množinou metód reprezentujúcich neterminály, pričom vloženie pravej strany pravidla odpovedajúceho neterminálu je simulované zavolaním metódy neterminálu. Odstránenie terminálu zo zásobníka je simulované porovnaním s identifikátorom tokena na vstupe. Vďaka tomu môže byť takmer celý analyzátor implementovaný pomocou konštruktov daného programovacieho jazyka.

Nevýhoda týchto analyzátorov je v zápise *LL(k)* gramatík, ktoré neumožňujú používať ľavú rekurziu. Tá musí byť z gramatiky odstránená, čo zvyčajne vedie k zneprehľadneniu výslednej gramatiky.

Medzi nástroje založené na *LL(k)* gramatikách patria napríklad:

Coco/R

Coco/R [21] je generátor analyzátorov, ktorý na základe predanej gramatiky vstupného jazyka vygeneruje skener, ktorý je implementovaný ako deterministický konečný automat, a rekurzívne zostupný *LL(1)* parser. *LL(1)* konflikty je možné riešiť multi-symbolovým výhľadom alebo sémantickou kontrolou (viac napríklad v [15]), a preto je trieda prijímaných gramatík *LL(k)* pre ľubovoľné k. To však zhoršuje prehľadnosť a čitateľnosť vygenerovaného analyzátora.

Implementácia je dostupná pre celú radu jazykov vrátane *C++*, *Java* a *C#*. Zdrojový kód analyzátora generovaný nástrojom *Coco/R* nepoužíva takmer žiadne nevhodné konštrukty, avšak k jeho prehľadnosti ešte nejaké veci chýbajú. Príkladom by mohlo byť používanie číselného kódu terminálu pri porovnávaní s výhľadom, ku ktorému je pripojený komentár obsahujúci tvar odpovedajúceho terminálu. To by bolo dobré nahradiť vhodnejším pomenovaním.

Grammatica

Grammatica [23] je generátor parserov pre jazyk *C#* a *Java* využívajúci *LL(k)* gramatiky zapísané v súbore s vlastným, pomerne jednoduchým formátom. Nástroj *Grammatica* generuje analyzátory s prehľadným zdrojovým kódom, ktorý je vhodne členený na logické celky a ľahko čitateľný. Vygenerovaný analyzátor však neimplementuje parser priamo použitím jazykových konštruktov, ale využíva predpočítanú dátovú štruktúru derivačného stromu. To neodpovedá našej predstave o tom ako by mal fungovať náš nástroj, ale radí ho to viac k nástroju *KindTrans* (viď neskôr). Zároveň mu to umožňuje jednoduchšie implementovať automatické zotavenie z chýb a podrobnejšie chybové správy.

JavaCC

JavaCC [24] je generátor parserov produkujúci zdrojový kód v jazyku *Java*. Vygenerovaný parser je štandardne *LL(1)* rekurzívne zostupný parser, avšak na niektorých miestach (podobne ako *Coco/R*) umožňuje využiť aj dlhší výhľad k rozlíšeniu nejasností. To však tiež zhoršuje jeho čitateľnosť a pochopiteľnosť. Lexikálna aj gramatická špecifikácia je uložená v jednom súbore a umožňuje používať regulárne výrazy priamo v špecifikácii gramatiky. Nástroj okrem toho poskytuje množstvo dodatočnej funkcionality súvisiacej napríklad s generovaním dokumentácie, hlásením chybových stavov alebo ladením a testovaním. V najnovšej verzii je pridaná možnosť generovania zdrojových kódov aj v *C/C++*.

LR(k) analyzátory

LR(k) analyzátory analyzujú vstup zľava doprava, pričom produkujú najpravejšiu deriváciu (viď definície 3.4 a 3.5). Symbol k označuje dĺžku potrebného výhľadu. Pri syntaktickej analýze využíva analyzátor zásobník a dve operácie - *posunutie* a *redukciu*.

Pri operácii posunutia je aktuálny token na vstupe vložený na zásobník a vstup sa posunie na nasledujúci token. Pri operácii redukcie je sekvencia terminálov a neterminál na vrchole zásobníka nahradená neterminálom na ľavej strane pravidla, ktorého pravá strana odpovedá odstránenej sekvencii. Na rozhodnutie o po-

užití operácie redukcie alebo posunutia sa využíva konečný automat, pričom problém nastáva, ak je v nejakom stave analyzátora možné vykonať obe operácie, prípadne je možné vykonať operáciu redukcie podľa viacerých pravidiel. V takom prípade je potrebné k rozhodnutiu využiť výhľad. Analýza končí v momente, kedy je na zásobníku len počiatočný neterminál a na vstupe už nie je žiadny token (v prípade, že musí byť spracovaný celý vstup).

LR analyzátory sú spravidla implementované použitím tabuľky[1]. Dodatočné úpravy a údržbu analyzátora však tento prístup neuľahčuje. Pre úplnosť úvadzame príklad generátoru takýchto analyzátrov.

Bison

Bison [20] je nástroj, ktorý pre deterministickú bezkontextovú gramatiku zapísanú v *BNF* notácii vygeneruje *LALR(1)* parser (verzia LR parseru, ktorá vyžaduje menšiu tabuľku a zachováva väčšinu použiteľných syntaktických konštruktov). *Bison* umožňuje manipulovať so sémantickým zásobníkom a špecifikovať sémantické akcie, ktoré môže užívateľ asociovať s pravidlami gramatiky.

Vygenerovaný zdrojový kód môže byť v jazyku *C*, *C++* alebo experimentálne v jazyku *Java*. Tento zdrojový kód však nie je jednoducho modifikovateľný a je pomerne komplikovaný na pochopenie. V kóde sa využíva rada konštruktov, ktoré by sa dali označiť ako nevhodné z hľadiska čitateľnosti. Jedná sa najmä o použitie konštruktu `goto:` a pomerne komplikovaných makier. To spôsobuje, že štruktúra vygenerovaného analyzátora nie je na prvý pohľad úplne jasná.

Bison zároveň používa ku generovaniu lexikálneho analyzátora externý nástroj *Flex* [25].

Prívetivé gramatiky

V tejto práci sa budeme zaoberať generovaním analyzátorov pre triedu k-prívetivých gramatík, ktoré sú podrobnejšie popísané v nasledujúcej kapitole. Ich výhodou oproti *LL(k)* gramatikám je, že umožňujú využiť ľavú rekurziu v zápise pravidiel neterminálov. Na prívetivých gramatikách je založených niekoľko už existujúcich nástrojov.

Predchodcom tejto práce je implementácia nástroja *KindCons* [8], ktorý slúži na generovanie analyzátorov v jazyku *C* a čiastočne aj nástroj *KindTrans* [9], ktorý predstavuje prívetivý translátor. Na tieto práce priamo nadväzuje implementácia nástroja ExTra [10], ktorý zavádza plne funkčnú rozšíriteľnosť a redukciu pravidiel gramatiky za behu.

Okrem toho existujú ďalšie implementácie v jazyku *Java*. Nástroj *JKindCons* [12] predstavuje implementáciu generátora prívetivých analyzátorov a nástroj *JKind* [11] predstavuje implementáciu prívetivého translátora. Oba nástroje sa v rôznych oblastiach snažia o vylepšenie pôvodnej implementácie nástrojov *KindCons* a *KindTrans*.

[1] Existujú aj práce zaoberajúce sa generovaním *LR* parserov s využitím rekurzívneho zostupu, napríklad [13]

Kapitola 3

Prívetivá analýza

3.1 Základné pojmy

V tejto časti sú uvedené základné pojmy z teórie jazykov a prekladačov, ktoré sú potrebné k popísaniu prívetivých gramatík a prívetivej analýzy.

Definícia 3.1:
(Prevzaté z [1], strana 15-16; preložené, upravené.)

Nech je Σ ľubovoľná konečná množina symbolov (abeceda). Konečnú postupnosť $w = a_1, \ldots, a_n$ $(a_i \in \Sigma)$ nazývame *slovo* nad abecedou Σ a budeme ju zapisovať $a_1 \ldots a_n$.

Počet všetkých symbolov slova w voláme dĺžka slova w a označujeme $|w|$. Slovo dĺžky 0 nazývame *prázdne* slovo a označujeme ho ε.

Množinu všetkých slov nad abecedou Σ označujeme Σ^* a množinu všetkých slov nad abecedou Σ bez prázdneho slova označujeme $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Nech x, y a $z \in \Sigma^*$, potom hovoríme, že:

1. xy je *zreťazenie* slov x a y.
2. x je *prefix* slova xy.
3. y je *sufix* slova xy.
4. x je *podslovo* slova yxz.

Ak je navyše $x \neq \varepsilon$, $x \neq y$ a x je prefix (sufix, podslovo) y, potom je x nazývané *vlastný* prefix (sufix, podslovo) y.

Definícia 3.2 (Gramatika):
(Prevzaté z [3], strana 119; preložené, upravené.)

Gramatikou nazývame štvoricu $G = (N, T, S, P)$, kde N a T sú disjunktné konečné abecedy, $S \in N$, P je konečná množina prepisovacích pravidiel tvaru $\alpha \to \beta$, kde $\alpha, \beta \in (N \cup T)^*$ a α obsahuje aspoň jeden symbol z N.

N sa nazýva množinou *neterminálov*, T sa nazýva množinou *terminálov* a S je *počiatočný symbol*.

Poznámka 3.3:
V práci sa používa nasledujúca konvencia:

- $a, b, c, \ldots,$ reprezentujú terminály.

- $A, B, C, \ldots$, reprezentujú neterminály.
- $\ldots, X, Y, Z$, reprezentujú terminály alebo neterminály.
- $\alpha, \beta, \gamma, \ldots$, reprezentujú slová zložené z terminálov aj neterminálov.
- $\ldots, x, y, z$, reprezentujú slová zložené z terminálov.

Definícia 3.4:
(Prevzaté z [2] strana 200 a [1] strana 86; preložené, upravené.)
Nech $G = (N, T, S, P)$ je gramatika a $\pi, \rho \in (N \cup T)^*$, potom hovoríme, že:

1. π *sa priamo prepíše na* ρ (značíme $\pi \Rightarrow_G \rho$), práve ak existujú slová $\alpha, \beta, \gamma, \delta \in (N \cup T)^*$ také, že $\pi = \gamma\alpha\delta, \rho = \gamma\beta\delta$ a $\alpha \to \beta$ je prepisovacie pravidlo z P.
2. π *sa prepíše na* ρ (značíme $\pi \Rightarrow_G^* \rho$), práve ak existuje postupnosť

$$\pi_1, \pi_2, \ldots, \pi_n \ (n \geq 1)$$

slov z $(N \cup T)^*$ takých, že

$$\pi = \pi_1 \Rightarrow_G \pi_2 \Rightarrow_G \cdots \Rightarrow_G \pi_n = \rho.$$

Postupnosť $\pi_1, \pi_2, \ldots, \pi_n$ sa nazýva *derivácia* (odvodenie) slova ρ zo slova π.
Ak je z kontextu jasné, že sa jedná o grammatiku G, môžeme označenie gramatiky vynechať a písať $\Rightarrow$, $\Rightarrow^*$.

Definícia 3.5:
(Prevzaté z [2] strana 201; preložené, upravené.)
Ak je v derivácii prepisovaný vždy najľavejší neterminál hovoríme o *najľavejšej derivácii* a značíme to $\Rightarrow_{Glm}^*$ ($\Rightarrow_{lm}^*$, ak je gramatika jasná z kontextu).
Ak je v derivácii prepisovaný vždy najpravejší neterminál hovoríme o *najpravejšej derivácii* a značíme to $\Rightarrow_{Grm}^*$ ($\Rightarrow_{rm}^*$).

Definícia 3.6:
(Prevzaté z [3], strana 119; preložené, upravené.)
Jazyk L(G) generovaný gramatikou G je definovaný následovne:

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

Jazyk $L(G)$ je teda tvorený všetkými slovami nad terminálnou abecedou, ktoré je možné odvodiť z počiatočného symbolu S.

Definícia 3.7 (Bezkontextová gramatika):
(Prevzaté z [3], strana 124; preložené, upravené.)
Gramatika $G = (N, T, S, P)$ sa nazýva *bezkontextová* (CFG), ak všetky pravidlá z P majú tvar $A \to \alpha$, kde $A \in N$ a $\alpha \in (N \cup T)^*$.

Definícia 3.8 (A-pravidlo):
(Prevzaté z [1] strana 150; preložené, upravené.)
A-pravidlo bezkontextovej gramatiky je pravidlo tvaru

$$A \to \alpha \ ,$$

kde $A \in N$ a $\alpha \in (N \cup T)^*$.

Definícia 3.9 (First):
(Prevzaté z [1], strana 300; preložené, upravené.)
Pre bezkontextovú gramatiku $G = (N, T, S, P)$, celé číslo k a $\alpha \in (N \cup T)^*$ definujeme:

$$First_G^k(\alpha) = \{x \mid \alpha \Rightarrow_{lm}^* x\beta \wedge |x| = k \vee \alpha \Rightarrow^* x \wedge |x| < k\}$$

Ak je z kontextu jasné, že sa jedná o gramatiku G, môžeme označenie gramatiky vynechať; ak je zároveň k = 1, píšeme *First(α)*.

Definícia 3.10 (Follow):
(Prevzaté z [1], strana 343; preložené, upravené.)
Pre bezkontextovú gramatiku $G = (N, T, S, P)$, celé číslo k a $\beta \in (N \cup T)^*$ definujeme:

$$Follow_G^k(\beta) = \{w \mid S \Rightarrow^* \alpha\beta\gamma \wedge w \in First_G^k(\gamma)\}$$

Ak je z kontextu jasné, že sa jedná o gramatiku G, môžeme označenie gramatiky vynechať; ak je zároveň k = 1, píšeme *Follow(β)*.

Definícia 3.11 (LL(k) gramatika):
(Prevzaté z [1], strana 336; preložené, upravené.)
Bezkontextová gramatika $G = (N, T, S, P)$ je *LL(k)*, ak pre každé dve najľavejšie derivácie

1. $S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow_{lm}^* wx$ a
2. $S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow_{lm}^* wy$

také, že $First_k(x) = First_k(y)$, platí, že $\beta = \gamma$.

Definícia 3.12 (Silná LL(k) gramatika):
(Prevzaté z [1], strana 343-344; preložené, upravené.)
Bezkontextová gramatika $G = (N, T, S, P)$ je *silná LL(k)*, pokiaľ platí: Ak $A \to \beta$ a $A \to \gamma$ sú rôzne A-pravidlá, tak

$$First_k(\beta Follow_k(A)) \cap First_k(\gamma Follow_k(A)) = \emptyset.$$

Definícia 3.13 (Translačná gramatika):
(Prevzaté z [5] strana 45; preložené, upravené.)
Translačná gramatika je pätica $G = (N, T, O, P, S)$, kde

1. N je konečná množina neterminálnych symbolov,
2. T je konečná množina symbolov, ktoré budeme nazývať *vstupné symboly*,
3. O je konečná množina symbolov, ktoré budeme nazývať *výstupné symboly*,
4. P je konečná množina pravidiel tvaru $A \to \alpha$, kde $A \in N, \alpha \in (N \cup T \cup O)^*$,
5. S je počiatočný symbol z N.

Pritom platí, že $T \cap D = \emptyset$ a $(T \cup D) \cap N = \emptyset$.

Definícia 3.14 (Sémantická translačná gramatika):
(Prevzaté z [7] strana 90; preložené, upravené.)
Sémantická translačná bezkontextová gramatika je šestica $G = (N, T, O, A, P, S)$, kde

1. N je konečná neprázdna množina neterminálnych symbolov,
2. T je konečná neprázdna množina vstupných symbolov,
3. O je konečná množina výstupných symbolov,
4. A je konečná množina *sémantických akcií*,
5. P je konečná množina pravidiel tvaru $A \to \alpha$, kde $A \in N$ a $\alpha \in (N \cup T \cup O \cup A)^*$,
6. S je počiatočný symbol z N.

Pritom platí, že N, T, A a O sú navzájom disjunktné.

Definícia 3.15:
(Prevzaté z [4], strana 221-222; preložené, upravené.)
Zásobníkový automat je sedmica

$$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

kde

1. Q je neprázdná konečná množina stavov,
2. Σ je neprázdná konečná vstupná abeceda,
3. Γ je neprázdná konečná zásobníková abeceda,
4. δ je zobrazenie $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ do konečných podmnožin $Q \times \Gamma^*$,
5. q_0 je počiatočný stav,
6. $Z_0 \in \Gamma$ je počiatočný zásobníkový symbol,
7. $F \subseteq Q$ je množina koncových stavov.

3.2 Prívetivé gramatiky

Teória uvedená vo zvyšku tejto kapitoly je prevzatá z prác [7] a [10] a je tu uvedená, pretože predstavuje základ pre vytváranie analyzátorov popisovaných v tejto práci.

Každé pravidlo ľubovoľnej bezkontextovej gramatiky môžeme zaradiť práve do jednej z nasledujúcich skupín:

1. pravidlá bez ľavej rekurzie (NLRP[1]),

[1]NLRP = Non-left-recursive production

2. pravidlá s priamou ľavou rekurziou (DLRP[2]),
3. pravidlá s nepriamou ľavou rekurziou (ILRP[3]),
4. pravidlá so skrytou ľavou rekurziou (HLRP[4]),
5. pravidlá so skrytou nepriamou ľavou rekurziou (HILRP[5]).

Jednotlivé skupiny definujeme následovne:

Definícia 3.16 (Produkcie bez ľavej rekurzie (NLRP)):

$$NLRP_G(A) = \{A \to \alpha \mid A \to \alpha \in P_G \wedge (\forall\beta)\alpha \not\Rightarrow_G^* A\beta\}$$
$$NLRP_G = \bigcup_{A \in N_G} NLRP_G(A)$$

Definícia 3.17 (Produkcie s priamou ľavou rekurziou (DLRP)):

$$DLRP_G(A) = \{A \to A\alpha \mid A \to A\alpha \in P_G \wedge \alpha \neq \varepsilon\}$$
$$DLRP_G = \bigcup_{A \in N_G} DLRP_G(A)$$

Definícia 3.18 (Produkcie s nepriamou ľavou rekurziou (ILRP)):

$$ILRP_G(A) = \left\{ \; A \to \alpha \;\middle|\; \begin{array}{l} A \to \alpha \in P_G \wedge (\exists\beta)\alpha \Rightarrow_G^+ A\beta \\ \wedge\; A \to \alpha \notin DLRP_G(A) \end{array} \right\}$$
$$ILRP_G = \bigcup_{A \in N_G} ILRP_G(A)$$

Definícia 3.19 (Produkcie so skrytou ľavou rekurziou (HLRP)):

$$HLRP_G(A) = \{A \to \alpha A\beta \mid A \to \alpha A\beta \in P_G \wedge \alpha \neq \varepsilon \wedge \alpha \Rightarrow_G^+ \varepsilon\}$$
$$HLRP_G = \bigcup_{A \in N_G} HLRP_G(A)$$

Definícia 3.20 (Produkcie so skrytou nepriamou ľavou rekurziou (HILRP)):

$$HILRP_G(A) = \left\{ \; A \to \alpha\beta \;\middle|\; \begin{array}{l} A \to \alpha\beta \in P_G \wedge \alpha \neq \varepsilon \wedge \alpha \Rightarrow_G^+ \varepsilon \;\wedge \\ (\exists\gamma)(\beta \to_G^+ A\gamma) \wedge (\forall\delta)(\beta \neq A\delta) \end{array} \right\}$$
$$HILRP_G = \bigcup_{A \in N_G} HILRP_G(A)$$

[2]DLRP = Directly left-recursive production
[3]ILRP = Indirectly left-recursive production
[4]HLRP = Hidden left-recursive production
[5]HILRP = Hidden indirectly left-recursive production

Definícia 3.21 (Follow mimo ľavej rekurzie):
Nech G je bezkontextová gramatika, ktorá obsahuje len pravidlá z NLRP a DLRP. Množina terminálnych slov dĺžky k, ktoré sa môžu objaviť priamo za daným neterminálom A mimo jeho ľavej rekurzie sa nazýva *následníci A mimo ľavej rekurzie* ($NLRF_k(A)$). Formálne:

$$NLRF_G^k(A) = \left\{ \begin{array}{l} a_1 \dots a_m \in \\ First_G^k(\beta \cdot Follow_G^k(B)) \end{array} \;\middle|\; \begin{array}{l} m \leq k \wedge B \to \alpha A \beta \in P_G \\ \wedge\, ((\alpha \not\Rightarrow_G^* \varepsilon) \vee (B \neq A)) \end{array} \right\}$$

Definícia 3.22 (Follow v ľavej rekurzii):
Nech G je bezkontextová gramatika, ktorá obsahuje len pravidlá z NLRP a DLRP. Množina terminálnych slov dĺžky k, ktoré sa môžu objaviť priamo za daným neterminálom A v ľavej rekurzii sa nazýva *následníci A v ľavej rekurzii* ($DLRF_k(A)$). Formálne:

$$DLRF_G^k(A) = \left\{ \begin{array}{l} a_1 \dots a_m \in \\ First_G^k(\alpha \cdot Follow_G^k(A)) \end{array} \;\middle|\; \begin{array}{l} m \leq k \wedge A \to A\alpha \in P_G \\ \wedge\, \alpha \neq \varepsilon \end{array} \right\}$$

Definícia 3.23 ((Silná) k-prívetivá gramatika):
Bezkontextová gramatika *G = (N, T, S, P)*, ktorá má len pravidlá bez ľavej rekurzie alebo s priamou ľavou rekurziou sa nazýva *(silná) k-prívetivá* (K(k)), ak pre každé $A \in N$ platia nasledujúce dve podmienky:

1. $DLRF_k(A) \cap NLRF_k(A) = \emptyset$;
2. pre každé dve pravidlá, ktoré sú buď obe z NLRP(A) ($A \to \alpha\beta$, $A \to \alpha\gamma$ a α je najdlhší spoločný prefix ich pravých strán) alebo obe z DLRP(A) ($A \to A\alpha\beta$, $A \to A\alpha\gamma$ a $A\alpha$ je najdlhší spoločný prefix ich pravých strán), platí
$$First_k(\beta \cdot Follow_k(A)) \cap First_k(\gamma \cdot Follow_k(A)) = \emptyset.$$

Definícia 3.24 (Prívetivá gramatika):
Bezkontextová gramatika G sa nazýva *prívetivá*, ak je k-prívetivá pre nejaké $k > 0$. Triedu prívetivých gramatík označujeme KG.

Definícia 3.25 (Slabá k-prívetivá gramatika):
Bezkontextová gramatika *G = (N, T, S, P)*, ktorá má len pravidlá bez ľavej rekurzie alebo s priamou ľavou rekurziu sa nazýva *slabá k-prívetivá* (WK(k)), ak pre každé $A \in N$ platia nasledujúce dve podmienky:

1. $DLRF_k(A) \cap NLRF_k(A) = \emptyset$;
2. ak existuje $w \in T^*$, $\delta \in (N \cup T)^*$ také, že $S \Rightarrow_{lm}^* wA\delta$, tak platí: pre každé dve pravidlá, ktoré sú buď obe z NLRP(A) ($A \to \alpha\beta$, $A \to \alpha\gamma$ a α je najdlhší spoločný prefix ich pravých strán) alebo obe z DLRP(A) ($A \to A\alpha\beta$, $A \to A\alpha\gamma$ a $A\alpha$ je najdlhší spoločný prefix ich pravých strán), že
$$First_k(\beta\delta) \cap First_k(\gamma\delta) = \emptyset.$$

3.3 Princípy prívetivej analýzy

Princípy prívetivej analýzy a konštrukcie dátových štruktúr, z ktorých budeme následovne generovať analyzátor ukážeme na príklade gramatiky aritmetického výrazu.

Príklad 3.26 (Gramatika jednoduchého aritmetického výrazu):

$$G = (N_G, T_G, P_G, S_G)$$

$$N_G = \{S,\ E,\ T,\ F\}$$
$$T_G = \{id,\ num,\ (,\ ),\ +,\ *\}$$
$$P_G = \left\{ \begin{array}{l} S \to E, \\ E \to E + T,\ E \to T, \\ T \to T * F,\ T \to F, \\ F \to id,\ F \to num,\ F \to (E) \end{array} \right\}$$
$$S_G = S$$

Na každú množinu pravidiel prívetivej gramatiky G sa môžeme pozerať ako na *les stromov pravidiel* [6]. Tento les je jednoducho preveditelný na dátovú štruktúru, z ktorej budeme generovať kód statického analyzátora prijímajúceho jazyk *L(G)*. V následujúcej časti popíšeme proces vytvorenia tejto štruktúry.

Pravidlá najskôr rozdelíme do skupín podľa neterminálu na ľavej strane pravidla a v rámci skupín ešte podľa toho, či pravidlo obsahuje ľavú rekurziu (ak pravá strana pravidla začína rovnakým neterminálom, aký je na ľavej strane pravidla). Výsledok pre gramatiku aritmetického výrazu je zobrazený na obrázku 3.1. Každý riadok obsahuje pravidlá s rovnakým neterminálom na ľavej strane, v ľavom stĺpci sú pravidlá bez ľavej rekurzie a v pravom stĺpci pravidlá s ľavou rekurziou.

Pravidlo bez ľavej rekurzie	Pravidlo s ľavou rekurziou
$S \to E$	
$E \to T$	$E \to E + T$
$T \to F$	$T \to T * F$
$F \to id$ $F \to num$ $F \to (E)$	

Obr. 3.1: Pravidlá rozdelené podľa neterminálu na ľavej strane a prítomnosti ľavej rekurzie.

V nasledujúcom kroku oddelíme neterminál z ľavej strany pravidla do samostatného stĺpca. Vďaka tomu nie je nutné uvádzať ľavé strany pravidiel, pretože to, aký neterminál tam patrí je jasné z riadku tabuľky. Rovnako je možné vynechať neterminál zo začiatku pravých strán rekurzívnych pravidiel (tiež je zrejmý z riadku tabuľky). Stav po tomto kroku je zobrazený na obrázku 3.2.

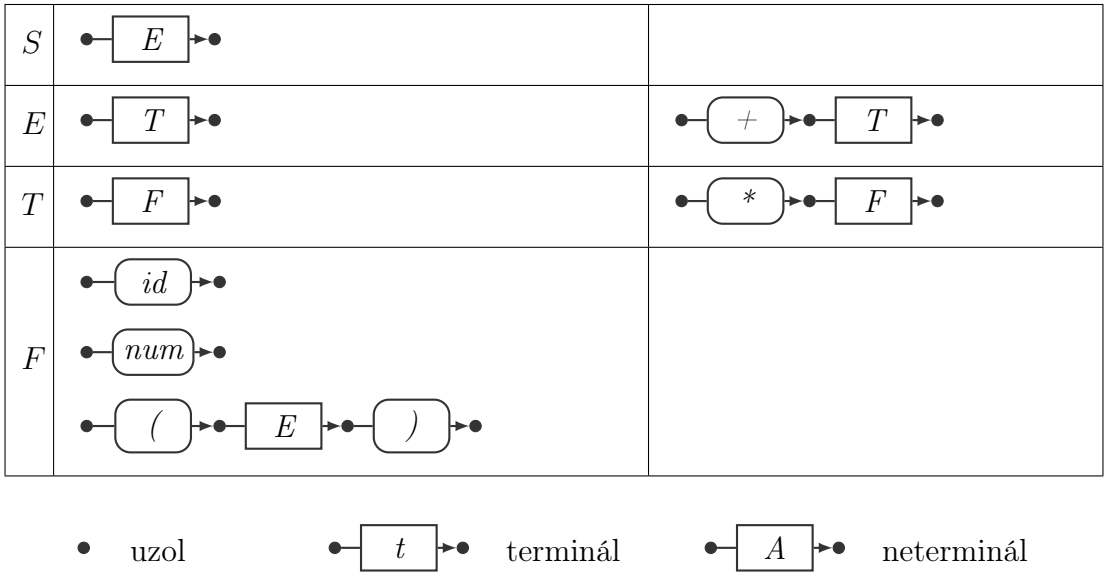
To isté je možné zapísať aj iným spôsobom. Každému pravidlu môžeme priradiť cestu v orientovanom grafe. Hrany na tejto ceste sú označené symbolmi (terminálmi a neterminálmi) daného pravidla. Tak ako na obrázku 3.2 neterminály

[6] V tejto práci budeme používať pojmy *strom pravidiel* a *produkčný strom* ako synonymá. Podobne budeme používať aj pojmy *pravidlo* a *produkcia*.

Neterminál	Pravá strana	
	pravidla bez ľavej rekurzie	pravidla s ľavou rekurziou
S	E	
E	T	$+T$
T	F	$*F$
F	id	
F	num	
F	(E)	

Obr. 3.2: Skrátený popis pravidiel rozdelených podľa neterminálu na ľavej strane pravidla a prítomnosti ľavej rekurzie.

v ľavej strane pravidla a na začiatku pravých strán pravidiel s ľavou rekurziou určené riadkom tabuľky, v ktorej sa nachádzajú a nie je nutné ich písať. Obrázok 3.3 ukazuje pravidlá aritmetiky jednoduchého výrazu zapísané ako cesty v orientovanom grafe.

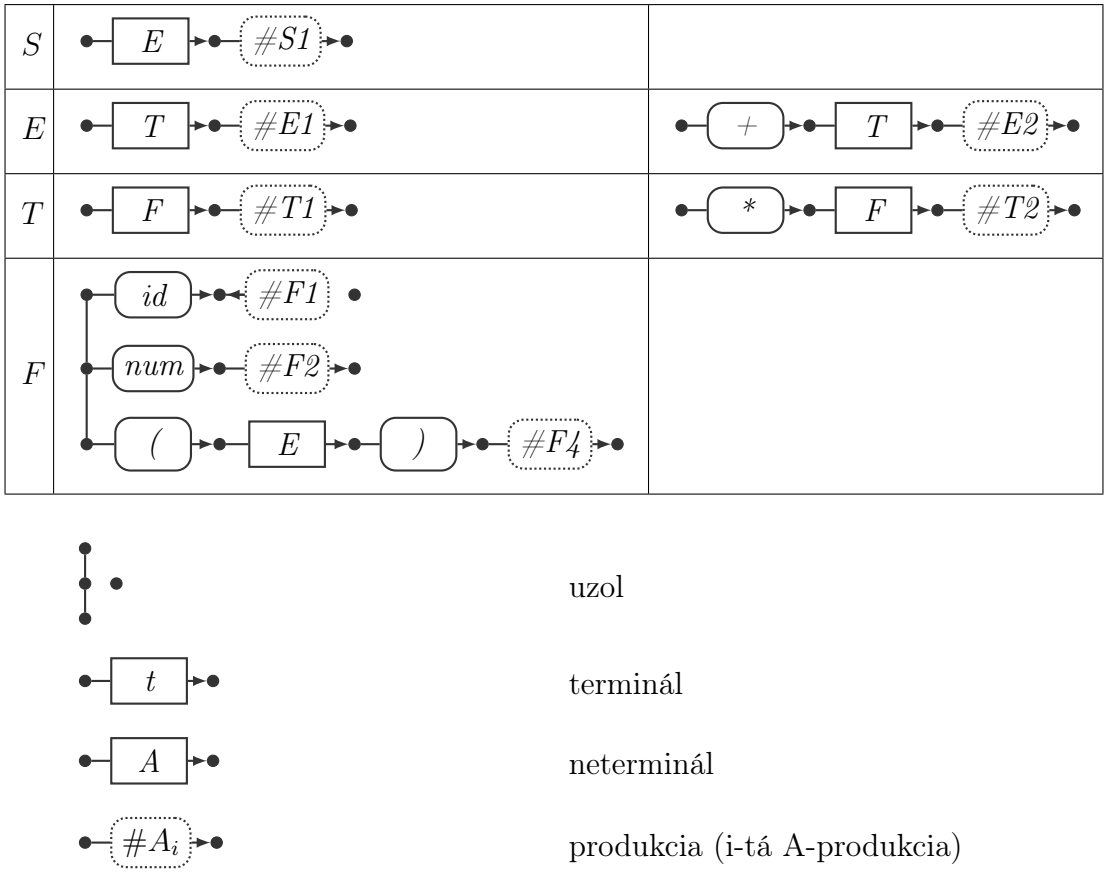


Obr. 3.3: Pravidlá gramatiky jednoduchého aritmetického výrazu zapísané ako cesty v orientovanom grafe.

Z praktických dôvodov je vhodné pridať na koniec každej cesty ešte jednu hranu, ktorá identifikuje dané pravidlo. Je to dôležité hlavne vtedy, keď je pravá strana nejakého pravidla prefixom pravej strany iného pravidla s rovnakou ľavou stranou. Hrana je označená neterminálom z ľavej strany pravidla a indexom pravidla (index môže byť určený napríklad poradím v akom boli pravidlá do grafu pridávané).

Nakoniec spojíme spoločné začiatky všetkých ciest pravidiel (spoločných prefixov pravých strán pravidiel) tak, aby vytvorili stromy - stromy pravidiel. Výsledok po identifikácii pravidiel a spojení ciest je zobrazený na obrázku 3.4.

Hrana nemusí byť ohodnotená len terminálom alebo neterminálom, v praxi použijeme napríklad ohodnotenie sémantickou akciou alebo výstupným terminálom.



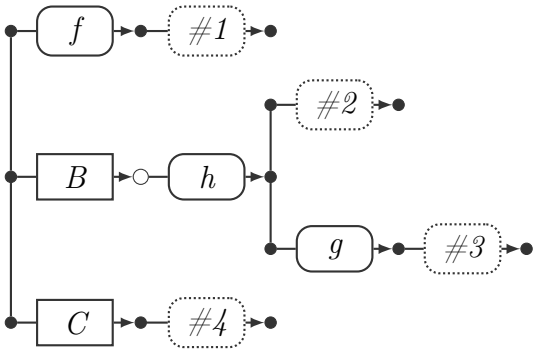
Obr. 3.4: Pravidlá gramatiky jednoduchého aritmetického výrazu zapísané ako cesty v strome pravidiel.

3.3.1 Použitie stromov pravidiel

Obrázky 3.4 a 3.5 môžu byť použité na definíciu parseru, implemenentovaného ako rekurzívny zostupný parser alebo zásobníkový automat. Symboly zásobníka a stavy automatu môžu byť implementované ako ukazovateľ alebo referencia na uzly (prípadne hrany) stromov pravidiel; určujú teda nejakú pozíciu v strome. Ukazovateľ na uzol bude znázornený čiernym krúžkom s bielym stredom (viď obrázok 3.5).

Pozícia v strome môže byť znázornená tiež pomocou tzv. *bodkovej notácie*. V bodkovej notácii je pravá strana pravidla rozdelená na dve časti pomocou symbolu bodky (.) - časť pred bodkou odpovedá už spracovaným symbolom a časť za bodkou odpovedá zatiaľ nespracovaným symbolom. Pre každé pravidlo $A \rightarrow \alpha\beta \in P$ existuje postupnosť bodkovaných pravidiel od "$A \rightarrow .\alpha\beta$" do "$A \rightarrow \alpha\beta.$".

V niektorých prípadoch môže jednej pozícii v strome pravidiel odpovedať viac rôznych bodkovaných pravidiel. To nastáva, ak je bodka na začiatku pravidiel alebo, ak má viac pravidiel s rovnakým neterminálom na ľavej strane rovnaký prefix pravých strán. Tieto bodkované pravidlá označíme ako ekvivalentné.



Pozícia vyznačená krúžkom odpovedá týmto bodkovaným pozíciam:

$$[A \to B.h]_{=G} = \{A \to B.h, A \to B.hg\}$$

Obr. 3.5: Pozícia v strome pravidiel v grafickom zobrazení.

Definícia 3.27 (Ekvivalencia bodkovaných pravidiel):
Pre ľubovoľnú bezkontextovú gramatiku *G = (N, T, S, P)* definujeme ekvivalenciu $=_G$ na bodkovaných pravidlách ako:

$$A \to \alpha.\beta =_G B \to \gamma.\delta \iff_{def} A = B \wedge \alpha = \gamma,$$

kde $A, B \in N$, $\alpha, \beta, \gamma, \delta \in (N \cup T)^*$, $A \to \alpha\beta, B \to \gamma\delta \in P$.

Triedu ekvivalencie $=_G$, ktorá obsahuje bodkované pravidlo $A \to \alpha.\beta$, označujeme $[A \to \alpha.\beta]_{=G}$, prípane len $[A \to \alpha.\beta]$.

Uvedené prípady tvoria triedy ekvivalencie $=_G$, ľubovoľné bodkované pravidlo môže byť reprezentant celej svojej triedy. To odpovedá grafickej notácii: každý uzol je reprezentovaný triedou ekvivalencie, ktorá je určená neterminálom na ľavej strane pravidla a pravou časťou pred bodkou. Časť pravidla pred bodkou udáva cestu v strome pravidiel k danej pozícii; ak sú uvedené ešte nejaké symboly, znamená to, že časť pred bodkou je prefixom nejakého existujúceho pravidla. Neterminál (označme ho A) určuje dva stromy pravidiel - jeden pre A-pravidlá bez ľavej rekurzie, druhý pre A-pravidlá s ľavou rekurziou, ak nejaké sú. Koreň stromu pre nejaké A-pravidlo s ľavou rekurziou môže byť označený $[A \to A.\alpha]$, kde $A \to A\alpha \in P$.

V nasledujúcej časti popíšeme chovanie analyzátora z pohľadu zásobníkového automatu.

Pri vlastnej analýze sa prechádzajú stromy pravidiel od koreňa stromu počiatočného symbolu a chovanie analyzátora závisí na ohodnotení jednotlivých hrán:

- Na začiatku analýzy mieri referencia na koreň stromu počiatočného symbolu, čo je v bodkovej notácii znázornené ako: $[S \to .\alpha]$, pre ľubovoľné pravidlo $S \to \alpha \in P$ a počiatočný symbol S.
- Analyzátor prechádza cez hranu s terminálom
  - zmení svoj stav (presunie referenciu na nasledujúci uzol v strome pravidiel),

  - prečíta terminál zo vstupu, musí sa zhodovať s terminálom z hrany, inak výpočet končí zamietnutím;

- Analyzátor prechádza cez hranu s neterminálom
  - zmení svoj stav (presunie referenciu na koreň stromu daného neterminálu s nerekurzívnymi pravidlami),
  - stav reprezentujúci koniec prechádzanej hrany vloží na zásobník,
  - vstup sa nemení;

- Analyzátor prechádza hranou s pravidlom
  - ak aktuálny výhľad vstupu odpovedá výhľadu neterminálu tohto pravidla do ľavej rekurzie (*DLRF*), zmení svoj stav (presunie referenciu na koreň stromu daného neterminálu s pravidlami s ľavou rekurziou),
  - ak aktuálny výhľad vstupu odpovedá výhľadu neterminálu tohto pravidla bez ľavej rekurzie (*NLRF*), končí spracovnie tohto neterminálu a zmení sa stav (referencia sa presunie na uzol, ktorým končí hrana ohodnotená neterminálom tohto pravidla, kvôli ktorému sme preskočili na koreň tohto stromu),
  - ak aktuálny výhľad vstupu neodpovedá *DLRF* ani *NLRF* a zásobník je prázdny, výpočet končí a vstup je prijatý,
  - inak výpočet končí a vstup je odmietnutý.

Prívetivý analyzátor je teda zásobníkový automat rozdelený na časti, ktoré odpovedajú jednotlivým neterminálom - ako keby bol tvorený rekurzívne volanými procedúrami. Najprv sa analyzuje nerekurzívna časť neterminálu podľa pravidiel bez ľavej rekurzie a potom, ak je to možné, podľa pravidiel s ľavou rekurziou.

Pre lepšie pochopenie si prácu prívetivého analyzátora ukážeme aj na jednoduchom príklade analýzy vstupného slova "3+2" podľa gramatiky jednoduchého aritmetického výrazu.

Príklad 3.28 (Práca prívetivého analyzátora):

Vstup	Stav	Zásobník
Začiatok výpočtu.		
.3+2	$[S \to .E]$	ε
Prechod k neterminálu E.		
.3+2	$[E \to .T]$	$[S \to E.]$
Prechod k neterminálu T.		
.3+2	$[T \to .F]$	$[S \to E.]$ $[E \to T.]$
Prechod k neterminálu F.		
.3+2	$[F \to .num]$	$[S \to E.]$ $[E \to T.]$ $[T \to F.]$
Prečítanie "3"zo vstupu.		

		$[S \to E.]$
		$[E \to T.]$
3.+2	$[F \to num.]$	$[T \to F.]$
Koniec rozvoja podľa pravidla $F \to num$. Nie je žiadne ľavorekurzívne F-pravidlo, takže rozvoj neterminálu F končí.		
		$[S \to E.]$
3.+2	$[T \to F.]$	$[E \to T.]$
Koniec rozvoja podľa pravidla $T \to F$. Teraz môžeme skúsiť, či je možné použiť nejaké ľavorekurzívne T-pravidlo. V tejto chvíli žiadne také nie je, takže rozvoj podľa neterminálu T končí.		
3.+2	$[E \to T.]$	$[S \to E.]$
Koniec rozvoja podľa pravidla $E \to T$. Teraz môžeme skúsiť, či je možné použiť nejaké ľavorekurzívne E-pravidlo. Také pravidlo je $E \to E + T$.		
3.+2	$[E \to E. + T]$	$[S \to E.]$
Prečítanie "+"zo vstupu.		
3+.2	$[E \to E + .T]$	$[S \to E.]$
Prechod k neterminálu T.		
		$[S \to E.]$
3+.2	$[T \to .F]$	$[E \to E + T.]$
Prechod k neterminálu F.		
		$[S \to E.]$
		$[E \to E + T.]$
3+.2	$[F \to .num]$	$[T \to F.]$
Prečítanie "2"zo vstupu.		
		$[S \to E.]$
		$[E \to E + T.]$
3+2.	$[F \to num.]$	$[T \to F.]$
Koniec rozvoja podľa pravidla $F \to num$. Nie je žiadne ľavorekurzívne F-pravidlo, takže rozvoj neterminálu F končí.		
		$[S \to E.]$
3+2.	$[T \to F.]$	$[E \to E + T.]$
Koniec rozvoja podľa pravidla $T \to F$. Nie je možné použiť žiadne ľavorekurzívne pravidlo $\Rightarrow$ dokončenie T.		
3+2.	$[E \to E + T.]$	$[S \to E.]$
Koniec rozvoja podľa pravidla $E \to E + T$. Nie je možné použiť žiadne ľavorekurzívne pravidlo $\Rightarrow$ dokončenie E.		
3+2.	$[S \to E.]$	ε
Koniec rozvoja podľa pravidla $S \to E$, bola dosiahnutá finálna pozícia $\Rightarrow$ koniec analýzy.		

Pri analýze je užitočné mať nejakú nápovedu, ktorá cesta v strome pravidiel môže viesť k cieľu, ak je dostupná viac ako jedna cesta. Táto nápoveda vychádza z výhľadových množín (*FIRST, FOLLOW, DLRF, NLRF*). Hodnoty nápovedy môžu byť zapísané v uzloch stromov; pri výpočte potom tieto hodnoty dovoľujú vybrať vetvu, ktorá odpovedá vstupu (alebo aspoň má šancu odpovedať vstupu) a cez ktorú bude výpočet pokračovať. Pre príklad gramatiky aritmetického výrazu stačia hodnoty výhľadových množín dĺžky 1. Navyše bol pridaný symbol *eoi* (end of input), symbolizujúci koniec vstupu. Vypočítané hodnoty množín *FIRST,*

FOLLOW, DLRF a *NLRF* pre gramatiku aritmetického výrazu sú uvedené v tabuľke na obrázku 3.6.

Neterminál	First	Follow	DLRF	NLRF
S	*num, id*, (	*eoi*		*eoi*
E	*num, id*, (	+,), *eoi*	+	), *eoi*
T	*num, id*, (	+, *,), *eoi*	*	+,), *eoi*
F	*num, id*, (	+, *,), *eoi*		+, *,), *eoi*

Obr. 3.6: Predpočítané hodnoty výhľadových množín dĺžky 1 pre gramatiku jednoduchého aritmetického výrazu.

Výpočet výhľadových množín a z nich odvodených výhľadov[7] jednotlivých hrán priblížime v nasledujúcej kapitole.

[7]Je nutné, aby čitateľ rozlišoval medzi pojmom *výhľadová množina* a *množina výhľadov*, prvý z týchto pojmov sa vzťahuje ku konkrétnemu neterminálu a predstavuje spoločné označenie pre množiny *First*, *Follow*, *NLRF* a *DLRF*, zatiaľ čo druhý označuje množinu terminálnych reťazcov vzťahujúcich sa ku konkrétnej hrane, ktorá je použitá počas syntaktickej analýzy.

Kapitola 4

Výhľadové množiny a výhľady

V tejto kapitole popíšeme vytváranie a použitie štruktúr, ktoré nám uľahčia prechod produkčnými stromami. V prvej časti sa bude jednať o spôsob výpočtu výhľadových množín, následne definujeme pojem výhľadu v rámci produkčného stromu a definujeme štruktúru, ktorá nám umožní tento výhľad efektívne používať. V závere kapitoly sa budeme venovať generovaniu zdrojového kódu s použitím predstavených štruktúr.

4.1 Náčrt výpočtu

Aby sme mohli vytvoriť výhľady pre jednotlivé hrany produkčných stromov, potrebujeme najskôr spočítať výhľadové množiny jednotlivých neterminálov. Pri výpočte týchto výhľadových množín budeme vychádzať z vlastností prívetivých gramatík a produkčných stromov predstavených v predchádzajúcej kapitole.

Výhľadové množiny používané v rámci generovania parseru vychádzajú zo štandardných výhľadových množín *First* a *Follow*. Podľa definície odpovedajú množiny $First(\alpha)$ a $Follow(\beta)$ množinám terminálnych reťazcov, získaných deriváciou odpovedajúcich symbolov (viď definície 3.9 a 3.10).

Zjednodušene povedané teda platí, že množina $First^k_G(\alpha)$ obsahuje všetky terminálne prefixy (prefixy obsahujúce iba terminálne symboly) do dĺžky k, ktoré vznikli deriváciou α pomocou pravidiel gramatiky G (prípadne kratšie ako k, ak už nie je možné ďalej derivovať).

Podobne množina $Follow^k_G(\beta)$ obsahuje všetky terminálne reťazce dĺžky k (prípadne kratšie, ak ďalší rozvoj nie je možný), ktoré sa môžu pri rozvoji počiatočného neterminálu pomocou pravidiel gramatiky G vyskytnúť za symbolom β.

Príklad 4.1 ilustruje výpočet množín $First$ a $Follow$ odpovedajúci definícii na príklade jednoduchej gramatiky. Pri výpočte množiny $First$ je neterminál A rozvíjaný pomocou pravidiel gramatiky pokiaľ prefix derivovaného reťazca neobsahuje potrebný počet terminálnych symbolov alebo už nie je možné tento reťazec ďalej derivovať.

Pri výpočte množiny $Follow$ je derivovaný počiatočný neterminál, pokiaľ vzniknutý reťazec neobsahuje neterminál, pre ktorý je množina $Follow$ počítaná (v tomto prípade je to neterminál B) a z reťazca, ktorý nasleduje za týmto neterminálom je spočítaná množina $First$ do požadovanej dĺžky.

Príklad 4.1 (Výpočet množín First^3(A) a Follow^3(B) podľa definície):
$N = \{S,\ A,\ B,\ C\}$
$T = \{a,\ b,\ c,\ d\}$
$$P = \left\{ \begin{array}{l} S \to A, \\ A \to Ba, A \to Cd, \\ B \to Cb, \\ C \to c \end{array} \right\}$$
$S = S$

Výpočet množiny $First$ do dĺžky 3 pre neterminál A:

$$\begin{array}{rcll} First^3(A) & = & First^3(Ba) & \cup \quad First^3(Cd) \\ & = & First^3(Cba) & \cup \quad First^3(cd) \\ & = & First^3(cba) & \cup \quad \{cd\} \\ & = & \{cba, cd\} & \end{array}$$

Výpočet množiny $Follow$ do dĺžky 3 pre neterminál B:
$S \Rightarrow^* Ba$
$Follow^3(B) = First(a) = \{a\}$

□

Tento postup funguje pre gramatiky s jednoduchými pravidlami. Problém však nastáva, ak sa v pravidlách gramatiky objaví rekurzia. Uvažujme napríklad, že by gramatika v príklade 4.1 obsahovala navyše pravidlo $A \to Ac$. Výpočet množiny $First^3(A)$ by potom obsahoval navyše túto vetvu:

$$\begin{array}{rcll} First^3(Ac) & = & First^3(Acc) & \cup \quad First^3(Bac) \quad \cup \quad First^3(Cbc) \\ & = & First^3(Accc) & \cup \quad \cdots \\ & = & First^3(Acccc) & \cup \quad \cdots \\ & = & \cdots & \end{array}$$

Jadro tohto problému teda spočíva v nekonečnom rozvoji neterminálu A pomocou rekurzívneho pravidla. Štandardne sa tento problém rieši odstraňovaním ľavej rekurzie z gramatiky. Naším cieľom však je, aby užívateľ mohol gramatiku definovať aj s pravidlami s ľavou rekurziou, pretože takto zapísaná gramatika je ľahšie čitateľná a prehľadnejšia. Predstavíme preto algoritmus založený na použití produkčných stromov predstavených v predchádzajúcej kapitole.

Algoritmus, ktorý predstavíme, je založený na jednoduchej úvahe, že každý rekurzívny rozvoj neterminálu musí byť ukončený nejakým rozvojom nerekurzívneho pravidla daného neterminálu.

Teda v prípade, že rozvíjame nejaký neterminál, môžeme najskôr spočítať rozvoj pomocou nerekurzívneho pravidla daného neterminálu a následne môžeme k získanému výsledku pripojiť možnosť, že už spočítaný rozvoj bol súčasťou nejakého rekurzívneho rozvoja. Pripájanie rekurzívnej časti je možné opakovať, čo slúži k simulácii rôznej hĺbky rekurzívneho zanorenia. Pre lepšie pochopenie si ukážeme spočítanie množiny $First$ pre neterminál A pre našu ukážkovú gramatiku (rozšírenú o nové rekurzívne pravidlo).

Príklad 4.2 (Výpočet množiny First^4(A) pripájaním):
Začneme rozvojom neterminálu A podľa jeho nerekurzívnych pravidiel:

$A \Rightarrow_{A \to Ba} Ba \Rightarrow_{B \to Cb} Cba \Rightarrow_{C \to c} cba$
$A \Rightarrow_{A \to Cd} Cd \Rightarrow_{C \to c} cd$

O týchto deriváciách môžeme nasledovne uvažovať ako o ukončení rekurzívneho rozvoja pomocou nerekurzívneho pravidla neterminálu A. K nim môžeme ľubovoľne veľakrát pripájať zvyšok pravej strany rekurzívneho pravidla podľa toho, akej hĺbke zanorenia má rozvoj odpovedať. Napríklad, pre hĺbku zanorenia 2 by to mohlo vyzerať nasledovne:

$cba \uparrow_{A \to Ac} = cba\mathbf{c}$
$cbac \uparrow_{A \to Ac} = cbac\mathbf{c}$

$cd \uparrow_{A \to Ac} = cd\mathbf{c}$
$cdc \uparrow_{A \to Ac} = cdc\mathbf{c}$

kde $\uparrow_{A \to Ac}$ značí predpoklad zanorenia v pravidle $A \to Ac$.

A teda napríklad $First^4(A) = \{cd, cba, cdc, cdcc, cbac\}$

□

Poznámka 4.3:
Pri rozvoji neterminálu podľa jeho nerekurzívnych pravidiel je nutné počítať s tým, že sa v týchto pravidlách objaví iný neterminál s rekurzívnymi pravidlami, pri ktorého rozvoji je nutné znova počítať jeho nerekurzívny rozvoj spolu so simuláciou rôznych hĺbok zanorenia do jeho rekurzie.

Tento postup je použiteľný aj pri počítaní množiny $Follow$ a navyše môžeme využiť nasledujúce pozorovanie, vďaka ktorému nebudeme musieť pri výpočte tejto množiny vždy vychádzať z derivácie počiatočného symbolu.

Pozorovanie 4.4:
Nech B je súčasťou nejakej derivácie počiatočného symbolu S, potom musí existovať postupnosť aplikácie pravidiel $p_0, p_1, \ldots, p_i$ taká, že B je na pravej strane pravidla p_i. $Follow(B)$ teda obsahuje terminálny reťazec vytvorený deriváciou symbolov nasledujúcich v pravidle p_i za symbolom B, ktorý je zreťazený s terminálnym reťazcom vytvoreným deriváciou symbolov, ktoré nasledujú v pravidle p_{i-1} za symbolom z ľavej strany pravidla p_i, ktorý je zreťazený s terminálnym reťazcom vytvoreným deriváciou symbolov, ktoré nasledujú v pravidle p_{i-2} za symbolom z ľavej strany pravidla p_{i-1}, atď., až kým sa nedostaneme späť k pravidlu s počiatočným symbolom S na ľavej strane.

Na základe tohto pozorovania môžeme definovať nasledujúcu množinu:

Definícia 4.5 (Množina následných reťazcov neterminálu):
Nech $G = (N, T, P, S)$, $A \in N$. Nech Q obsahuje všetky pravidlá z P, ktoré majú na svojej pravej strane neterminál A. Potom pre neterminál A definujeme *množinu následných reťazcov* $\#A$ nasledovne:

$\#A = \{(\gamma, i) \mid \alpha \to \beta A \gamma \in Q\}$,

kde i je identifikátor množiny následných reťazcov neterminálu α.

Poznámka 4.6:
Kvôli prehľadnosti budeme jednotlivé prvky množín následných reťazcov (γ, i) zapisovať v tvare γi, teda napríklad prvok *(+T, #E)* zapíšeme ako *+T#E*.

Na ilustráciu tohoto postupu si v príklade 4.7 predvedieme postup výpočtu množiny *Follow* neterminálu C do dĺžky 3 pre našu jednoduchú ukážkovú gramatiku, ktorú pre lepšiu prehľadnosť znova uvádzame v príklade.

K výpočtu budeme potrebovať pre každý neterminál spočítať množinu následných reťazcov daného neterminálu. Tú môžeme jednoducho vytvoriť podľa definície 4.5, prehľadaním pravých strán pravidiel gramatiky a pri výskyte hľadaného neterminálu si poznamenať zvyšné symboly pravej strany tohto pravidla spolu s identifikátorom množiny následných reťazcov neterminálu z ľavej strany pravidla, z ktorého reťazec pochádza.

Príklad 4.7 (Výpočet množiny Follow^3(C) použitím množiny následných reťazcov):
$N = \{S,\ A,\ B,\ C\}$
$T = \{a,\ b,\ c,\ d\}$
$$P = \left\{ \begin{array}{l} S \to A, \\ A \to Ba, A \to Cd, A \to Ac \\ B \to Cb, \\ C \to c \end{array} \right\}$$
$S = S$

Najskôr spočítame všetky množiny následných reťazcov:
$\#S = \{\}$
$\#A = \{\#S, c\#A\}$
$\#B = \{a\#A\}$
$\#C = \{d\#A, b\#B\}$

Následne môžeme uplatniť našu úvahu $Follow(C) = First(\#C)$:

$$\begin{aligned} Follow^3(C) &= First^3(\#C) \\ &= First^3(d\#A) \cup First^3(b\#B) \\ &= d \cdot First^3(\#A) \cup b \cdot First^3(\#B) \\ &= d \cdot First^3(\#S) \cup d \cdot First^3(c\#A) \cup b \cdot First^3(a\#A) \\ &= d \cup dc \cdot First^3(\#S) \cup dc \cdot First^3(c\#A) \\ &\quad \cup ba \cdot First^3(\#S) \cup ba \cdot First^3(c\#A) \\ &= d \cup dc \cup dcc \cdot First^3(\#S) \cup dcc \cdot First^3(c\#A) \\ &\quad \cup ba \cup bac \cdot First^3(\#S) \cup bac \cdot First^3(c\#A) \\ &= \{d, dc, dcc, ba, bac\} \end{aligned}$$

□

Algoritmus

Na základe predchádzajúcich úvah môžeme pomerne priamočiaro vytvoriť algoritmus, ktorý nám dovolí spočítať výhľadové množiny pre prívetivú gramatiku použitím produkčných stromov. Predtým však ešte popíšeme dátové štruktúry, ktoré algoritmus používa.

Node
Node predstavuje uzol v produkčnom strome. V tomto uzle je udržiavaný zoznam hrán, ktoré z daného uzla vedú. Okrem toho ďalej obsahuje odkaz na koreň výhľadového stromu (definícia 4.13) pre skupinu hrán z tohto zoznamu.

Edge
Edge predstavuje hranu v produkčnom strome. Hrana je ohodnotená symbolom, ktorého prechod reprezentuje (v implementácii je tento symbol identifikovaný dvojicou zloženou z typu symbolu a jeho jednoznačného kódu).

Type
Type predstavuje typ hrany v produkčnom strome. Typ hrany závisí na symbole, ktorého prechod hrana reprezentuje. Tento symbol môže byť jeden z nasledujúcich[1]:

1. terminal - odpovedajúci symbol je terminál,
2. nonterminal - odpovedajúci symbol je neterminál,
3. production - hrana s týmto typom označuje koniec pravidla gramatiky.

Nonterminal
Nonterminal predstavuje neterminál gramatiky. Pre popis algoritmu výpočtu výhľadov je nutné vedieť, že táto štruktúra obsahuje ukazovatele na korene produkčných stromov pre rekurzívne a nerekurzívne pravidlá daného neterminálu. Algoritmus má prístup k zoznamu týchto neterminálov, ktorý budeme označovať *nterms*.

Okrem ukazateľov na korene rekurzívneho a nerekurzívneho stromu obsahuje táto štruktúra ešte aj ukazateľ na koreň tzv. stromu následovníkov. Strom následovníkov je významovo podobný množine následných reťazcov. Koreň tohto stromu obsahuje všetky hrany, ktoré v rekurzívnych a nerekurzívnych stromoch nasledujú po hrane odpovedajúcej danému neterminálu. Je nutné si uvedomiť, že tento strom je množine následných reťazcov len podobný a to preto, že v rekurzívnych stromoch je vynechaná prvá hrana, ktorá by odpovedala rekurzii daného neterminálu. A tak hrana, ktorá by za ňou nasledovala, nie je v strome následovníkov odpovedajúceho neterminálu, zatiaľ čo následný reťazec by bol v množine následných reťazcov pre tento neterminál. Tento fakt priamo využijeme pri výpočte množiny *NLRF*.

Okrem týchto štruktúr algoritmus využíva ešte dva druhy zásobníkov. Prvý zásobník uchováva aktuálny medzivýsledok terminálneho reťazca, a teda sú do neho ukladané kódy terminálnych symbolov. Tento zásobník budeme označovať *TS*. Druhý zásobník slúži na odkladanie uzlov k neskoršiemu rozvoju a budeme ho označovať *NS*.

Ďalej algoritmus využíva dve tabuľky indexované počtom neterminálov a požadovanou dĺkou výhľadu. Tieto tabuľky budeme označovať *vFi* a *vFo* a ich presné využitie popíšeme neskôr.

Jadro algoritmu je implementované následovnou rekurzívnou procedúrou:

[1] V skutočnosti typ hrany môže mať ešte hodnoty `outterminal` a `action`, tieto hodnoty však nie sú pre výpočet výhľadu podstatné, a preto sú opomenuté.

```
makeView(int depth, Node& start, Stack& TS, Stack& NS, bool follow){
  if (depth <= 0) {
    save(TS);
    return;
  }

  foreach(Edge e in start.edges) {
    switch (e.type) {
      case Type::terminal:
        TS.push(e.code);
        makeView(depth-1, e.next, TS, NS, follow);
        TS.pop();
      break;

      case Type::nonterminal:
        if (vFi[e.code][depth])
          throw error_GrammarNotKind;
        vFi[e.code][depth] = true;

        NS.push(e.next);
        makeView(depth, nterms[e.code].nRecRoot,
            TS, NS, follow);
        vFi[e.code][depth] = false;
        NS.pop();
      break;

      case Type::production:
        if (vFo[e.code][depth]) {
          save(TS);
          return;
        }
        vFo[e.code][depth] = true;

        if(!NS.empty) {
          nsc = NS.copy;
          next = nsc.top();
          nsc.pop();
          makeView(depth, next, TS, nsc, follow);
        }
        else if (follow && !nterms[e.code].followRoot.empty)  {
          makeView(depth, nterms[e.code].followRoot,
              TS, NS, follow);
        }
        else {
          save(TS);
        }

        if (!nterms[e.code].recRoot.empty) {
          makeView(depth, nterms[e.code].recRoot,
              TS, NS, follow);
        }
        vFo[e.code][depth] = false;
      break;
    }
  }
```

```
56 }
```

Listing 4.1: Náčrt metódy *makeView* predstavujúcej jadro výpočtu výhľadových množín.

Tento algoritmus je možné rozdeliť na dve časti. Prvá časť je reprezentovaná riadkami 2 – 5 a slúži ako ukončovacia podmienka. Jej úlohou je overiť, či sa už podarilo dosiahnuť požadovanú dĺžku terminálneho reťazca a v kladnom prípade uložiť výsledok a ukončiť rekurziu. Dĺžka terminálneho reťazca, ktorú treba ešte dopočítať, je procedúre predávaná pomocou argumentu *depth*.

Druhá časť je reprezentovaná riadkami 7 – 56. Táto časť má za úlohu postupne spracovať všetky hrany uložené v uzle *start*. Spôsob spracovania hrany závisí na type symbolu, ktorý daná hrana predstavuje a ktorý spadá do jedného z nasledujúcich troch prípadov:

1. Hrana reprezentuje terminál:

   V tomto prípade je kód odpovedajúceho terminálu uložený na zásobník a je znova zavolaná procedúra *makeView*, ktorej je predaná dekrementovaná požadovaná dĺžka terminálneho reťazca (pretože sme práve do medzivýsledku pridali jeden terminálny symbol) a ako štartovný uzol je predaný uzol na ktorý ukazuje daná hrana. Po vynorení sa z tohto volania je nájdený terminálny symbol zo zásobníka odstránený.

2. Hrana reprezentuje neterminál:

   Tento prípad odpovedá prechodu neterminálom, ktorý je potrebné derivovať. Derivácia spočíva v zavolaní procedúry *makeView* na koreň nerekurzívneho stromu neterminálu odpovedajúceho neterminálu danej hrany. Keďže nebol žiadny terminál pridaný do medzivýsledku, je predaná rovnaká požadovaná dĺžka terminálneho reťazca.

   Pred zavolaním samotnej procedúry je ďalej nutné uložiť na zásobník uzol, na ktorý daná hrana ukazuje, aby ním mohlo byť v prípade potreby naviazané na výpočet po ukončení derivácie neterminálu reprezentovaného danou hranou. Po vynorení sa z tohto volania je znova tento uzol zo zásobníka odstránený.

   Okrem vyššie popísaného je v tejto časti ešte kontrola a nastavenie príznaku v tabuľke *vFi*. Tento príznak slúži k detekcii zacyklenia pri prechode hranou reprezentujúcou neterminál. Presný význam tohto kroku popíšeme v nasledujúcej sekcii.

3. Hrana reprezentuje koniec pravidla:

   Tento prípad značí, že bol dosiahnutý koniec nejakého pravidla v strome pravidiel, čo je ekvivalent ukončenia rozvoja nejakého neterminálu. Neterminál z ľavej strany tohto pravidla je možné určiť podľa kódu uloženého v danej hrane. Pri ukončení rozvoja neterminálu je možné pokračovať troma smermi na základe aktuálneho stavu výpočtu, ktorý je daný obsahom zásobníka odložených uzlov (*NS*), príznakom *follow* spolu s obsahom stromu následovníkov neterminálu odpovedajúceho danej hrane a obsahom rekurzívneho stromu neterminálu odpovedajúceho danej hrane.

V prípade, že zásobník odložených uzlov nie je prázdny, bol aktuálne rozvíjaný neterminál súčasťou nejakého pravidla a rozvoj môže pokračovať symbolom, ktorý nasledoval za týmto neterminálom. Môžeme teda zavolať procedúru *makeView* na uzol, ktorý sa nachádza na vrchole zásobníka odložených uzlov. Pretože pri spracovaní hrany ukončenia pravidla je možné pokračovať viacerými smermi zároveň, je pri tomto volaní vytvorená kópia zásobníka *NS*, ktorá je predaná procedúre.

Ak nie je splnená predchádzajúca podmienka, ale je nastavený príznak *follow* a strom následovníkov pre neterminál odpovedajúci tejto hrane nie je prázdny, je možné pokračovať zavolaním procedúry *makeView* na koreň stromu následovníkov. Táto vetva výpočtu odpovedá úvahe o výpočte množiny *Follow*, ktorú sme uviedli na strane 24, a teda je použitá len v prípade, že je počítaná niektorá forma *Follow* množiny (*NLRF* alebo *DLRF*).

V prípade, že predchádzajúce dve podmienky nie sú splnené, je medzivýsledok prehlásený za výsledok a uložený.

Následne sa musí ešte vyriešiť varianta, že je ukončovaný rozvoj neterminálu, ktorý bol súčasťou svojho rekurzívneho rozvoja, a preto je v prípade, že má neterminál odpovedajúci danej hrane neprázdny rekurzívny strom, je zavolaná procedúra *makeView* na koreň rekurzívneho stromu daného neterminálu.

Okrem vyššie popísaného je v tejto časti ešte kontrola a nastavenie príznaku v tabuľke *vFo*. Tento príznak slúži k detekcii zacyklenia pri prechode hranou reprezentujúcou koniec pravidla. Presný význam tohto kroku popíšeme v nasledujúcej sekcii.

Ako je vidieť, algoritmus simuluje deriváciu s použitím úvah, ktoré sme uviedli na začiatku tejto kapitoly. K úplnosti ostáva ukázať, že predstavená metóda pre každý vstup skončí v konečnom počte krokov.

Pri každom zanorení je prevádzaná kontrola zacyklenia pri prechode neterminálnou hranou a hranou pre koniec pravidla. Pri tomto prechode je v odpovedajúcej tabuľke poznačená aktuálna dĺžka spočítaného výhľadu a odpovedajúca hrana. Keďže počet neterminálov a pravidiel je konečný, nie je možné, aby bol do týchto tabuliek poznamenávaný vždy iný záznam, a preto sa v konečnom počte krokov musí nájsť už existujúci záznam a výpočet sa preruší (samozrejme len v prípade, že sa neprerušil skôr nájdením odpovedajúceho rozvoja).

Prechodom cez terminálnu hranu sa vždy znižuje dĺžka výhľadu, ktorú je potrebné dopočítať, a preto je rozvoj pomocou týchto hrán ohraničený ukončovacou podmienkou na začiatku metódy. Všetky tri vetvy tejto metódy sú teda ošetrené. Keďže počet hrán vedúcich z jedného uzlu je tiež konečný musí sa zastaviť aj cyklus, v ktorom dochádza k vetveniu výpočtu. Na základe týchto vlastností je možné povedať, že sa výpočet metódy vždy zastaví.

V nasledujúcej časti uvedieme ako je táto metóda použitá k výpočtu konkrétnych výhľadových množín *First*, *NLRF* a *DLRF* pre jednotlivé neterminály.

4.2 Výpočet výhľadových množín

First

Množinu $First^k(A)$ je možné spočítať derivovaním neterminálu A, pokiaľ získaný reťazec nemá ako prefix terminálny reťazec dĺžky k alebo už nie je možné získaný reťazec ďalej derivovať. Toto je možné simulovať použitím procedúry predstavenej v minulej sekcii.

Tejto procedúre je predaná požadovaná dĺžka k a koreň nerekurzívneho stromu daného neterminálu, čo zaručí, že najprv bude vykonanný rozvoj nerekurzívneho pravidla daného terminálu a následne sa metóda pokúsi simulovať pridanie časti reprezentovanej rekurzívnymi pravidlami tohto neterminálu. Výpočet teda odpovedá postupu, ktorý sme uviedli v minulej sekcii. Zároveň treba dodať, že pri výpočte množiny *First* sa nevyužíva strom následovníkov, a preto nie je nastavený príznak *follow*.

Pri tomto výpočte je však nutné prevádzať kontrolu zacyklenia. K zacykleniu procedúry môže dôjsť v prípade, že v nejakej vetve výpočtu sa procedúra pokúsi viackrát rozvíjať rovnaký neterminál bez toho, aby medzi týmito pokusmi pridala nejaký symbol k medzivýsledku.

Táto kontrola je v algoritme implementovaná pomocou tabuľky príznakov *vFi*. Počas výpočtu si do tejto tabuľky zaznamenávame, ktorý neterminál sme v danej vetve už rozvíjali a pri akej dĺžke medzivýsledku k tomuto rozvoju dochádzalo. Z týchto informácii už prípadný cyklus jednoducho detekujeme a môžeme výpočet prerušiť.

Hodnoty množín *First* jednotlivých neterminálov predstavujú medzivýsledok, ktorý nám pomôže pri zostavovaní celkového výhľadu pre hranu v produkčných stromoch.

NLRF a DLRF

Množiny *NLRF* a *DLRF* sa počítajú rovnakým algoritmom ako množina *Follow*. Líšia sa len tým, z ktorých hrán budeme vychádzať. Zatiaľ čo pri výpočte *Follow(A)* by boli použité všetky hrany repezentujúce symbol za neterminínalom A v nejakom pravidle, budú k výpočtu *DLRF(A)* použité iba hrany nasledujúce priamo za ľavou rekurziou neterminálu A (to sú presne hrany vedúce z koreňa rekurzívneho stromu neterminálu A) a k výpočtu *NLRF(A)* všetky hrany v produkčných stromoch za hranou odpovedajúcou neterminálu A (pri ľavej rekurzii je prvá hrana odpovedajúca rekurzívnemu neterminálu vynechaná). Hrany pre výpočet *NLRF* sú predpočítané pred spustním samotného výpočtu a uložené v koreni stromu následovníkov.

Pri tomto výpočte je podobne ako pri výpočte množiny *First* nutné detekovať zacyklenie. Okrem možnosti zacyklenia pri rozvoji neterminálu, existuje aj možnosť zacyklenia pri nasledovaní hrán v strome následovníkov. Môže k tomu dôjsť napríklad v situácii, keď v gramatike existuje pravidlo tvaru $A \rightarrow \alpha A$. Potom bude strom následovníkov obsahovať hranu, ktorá reprezentuje ukončenie pravidla neterminálu A. Táto hrana bude pri výpočte priamo spracovaná v časti ukončujúcej rozvoj neterminálu (Type::production), kde však bude znovu zavolané jej spracovanie zo stromu následovníkov bez toho, aby bolo niečo pridané k medzivýsledku.

Tento druh zacyklenia detekujeme pomocou tabuľky *vFo*, ktorá funguje rovnako ako tabuľka *vFi*. Pri detekcii tohto zacyklenia vytvoríme uzáver na aktuálnom medzivýsledku a danú vetvu výpočtu prerušíme.

Hodnoty množiny *DLRF* a *NLRF* spolu s hodnotami množiny *First* predstavujú medzivýsledok, ktorý nám pomôže pri zostavovaní celkového výhľadu pre hranu v produkčných stromoch.

Pretože prívetivé gramatiky obsahujú len pravidlá z NLRP a DLRP je jednoduché nahliadnuť, že pre každý neterminál platí:

$$Follow(A) = NLRF(A) \cup DLRF(A)$$

Poznámka 4.8:
Z technických dôvodov je pred začiatkom výpočtu výhľadových množín ku každej definícii gramatiky pridané nasledujúce pravidlo:

$$S' \rightarrow S\$,$$

kde:

S	je počiatočný neterminál gramatiky,
S'	je neterminál, ktorý bude počas výpočtu považovaný za počiatočný neterminál gramatiky,
\$	je implicitne definovaný terminál predstavujúci koniec vstupu.

Toto pravidlo zaručí, že pri analýze vstupného textu bude vždy ako posledný očakávaný terminál predstavujúci koniec vstupu (a teda, že celý vstupný text odpovedá danej gramatike). V rámci vygenerovaného analyzátora sa však s neterminálom S' vôbec nepočíta.

4.3 Spracovanie výhľadov

Spočítané výhľadové množiny zatiaľ predstavujú len medzivýsledok na ceste k plnohodnotným výhľadom. V nasledujúcej časti zavedieme definícu výhľadu a popíšeme spôsob, ako z výhľadových množín jednotlivých neterminálov vytvoriť množinu výhľadov pre jednotlivé hrany produkčných stromov.

Definícia 4.9 (Výhľad, výhľadové slovo, identifikátor hrany):
Výhľadom nazveme dvojicu $v = (s, b)$, kde $s \in \Sigma^+$ je *výhľadové slovo*, ktoré sa môže počas syntaktickej analýzy vyskytnúť na začiatku nespracovanej časti vstupného textu a b je *identifikátor hrany* v produkčnom strome, ktorou syntaktický analyzátor práve prechádza.

Identifikátorom hrany môže byť napríklad referencia alebo index hrany v produkčnom strome. Pre množinu výhľadov V zároveň definujeme množinu výhľadových slov $S = \{s \mid s \in v \wedge v \in V\}$ a množinu identifikátorov $B = \{b \mid b \in v \wedge v \in V\}$.

Výhľadom hrany e_n nazveme množinu výhľadov, ktorých identifikátor hrany je zhodný s identifikátorom hrany e_n.

Prvým dôležitým aspektom, ktorý využijeme pri výpočte výhľadu hrany, je vzťah medzi typom hrany produkčného stromu a jej odpovedajúcou výhľadovou množinou:

1. V prípade, že hrana v produkčnom strome odpovedá nejakému neterminálu A, bude rozvoj tejto hrany pri prechode produkčným stromom odpovedať rozvoju neterminálu A, podľa nejakého pravidla, ktoré má tento neterminál na ľavej strane. To znamená, že odpovedajúca výhľadová množina bude práve *First(A)*.

2. Ak hrana v produkčnom strome odpovedá nejakému terminálu t, bude rozvoj tejto hrany pri prechode produkčným stromom triviálny a bude odpovedať množine *First(t) = {t}*.

3. Nakoniec, ak hrana v produkčnom strome odpovedá ukončeniu nejakého pravidla, bude rozvoj tejto hrany odpovedať rozvoju reťazca nasledujúceho za neterminálom (povedzme A) z ľavej strany tohto pravidla v nejakej derivácii počiatočného symbolu. To odpovedá práve množine *Follow(A)*.

Poznámka 4.10:
Kvôli prehľadnosti môžeme v zápise výhľadových množín namiesto neterminálu či terminálu použiť označenie hrany. Teda, v prípade, že hrana e odpovedá nejakému neterminálu A (prípadne terminálu t), zapíšeme odpovedajúcu výhľadovú množinu ako $First(e)$. Rovnako v prípade, že hrana odpovedá ukončeniu nejakého pravidla, zapíšeme výhľadovú množinu tejto hrany ako $Follow(e)$.

Poznámka 4.11:
Produkčné stromy môžu obsahovať aj hrany iných typov, napríklad hrany odpovedajúce výstupným terminálom alebo sémantickým akciám. Tieto hrany však neovplyvňujú výhľad, a preto ich pri počítaní výhľadu nebudeme uvažovať.

Druhým dôležitým aspektom, ktorý je nutné brať do úvahy, je požadovaná dĺžka výhľadového slova. Ak by sme jednoducho priradili každej hrane odpovedajúcu výhľadovú množinu a tú prehlásili za výhľad tejto hrany, mohlo by dôjsť k situácii, že niektoré prvky tejto množiny nebudú dosahovať požadovanú dĺžku alebo ju prípadne budú presahovať. Tento problém je potrebné samostatne ošetriť.

Riešením tohto problému je pri vytváraní výhľadu pre hranu uvažovať aj výhľady hrán, ktoré nasledujú za danou hranou na ceste k listom produkčného stromu. Inými slovami, môžeme namiesto výhľadu hrany uvažovať o výhľade cesty a to nasledujúcim spôsobom:

Nech e_n je hrana v produkčnom strome, pre ktorú počítame výhľad a

$$e_n, e_{n+1}, e_{n+2}, \ldots, e_{n+i}$$

je nejaká cesta do listového uzlu produkčného stromu. Potom výhľadom cesty nazveme množinu:

$$First(e_n) \cdot First(e_{n+1}) \cdot \ldots \cdot First(e_{n+i-1}) \cdot Follow(e_{n+i})$$

V prípade, že niektorý prvok tejto množiny presahuje požadovanú dĺžku výhľadu, stačí odstrániť nadbytočné terminálne symboly tak, aby ostal len prefix správnej veľkosti.

Keďže hranou e_n môžu začínať cesty do viacerých listových uzlov, je nutné do výhľadu tejto hrany zahrnúť výhľady všetkých týchto ciest.

Na základe týchto vlastností je už jednoduché vytvoriť výhľady pre všetky hrany v produkčných stromoch. Výhľad hrany však ešte nepredstavuje štruktúru, ktorá by umožňovala efektívny výber hrany, ktorou sa má pokračovať pri syntaktickej analýze. Takúto štruktúru a jej použitie predstavíme v nasledujúcej časti tejto sekcie.

Definícia 4.12 (Stromová reprezentácia množiny reťazcov):
Nech S je neprázdna množina reťazcov nad Σ^+ taká, že žiadne $s \in S$ nie je prefixom iného $s' \in S$. Strom $T = (V, E)$ nazveme *stromovou reprezentáciou množiny reťazcov* z S, ak platí:

1. existuje ohodnotenie hrán $l : E \to \Sigma$ také, že zreťazením ohodnotení pozdĺž cesty z koreňa do listu vznikne slovo z S (hovoríme, že daná cesta reprezentuje dané slovo),

2. každá cesta z koreňa do listu reprezentuje práve jedno slovo z S a existuje bijekcia medzi cestami z koreňa do listov a slovami z S.

Definícia 4.13 (Výhľadový strom):
(Inšpirované výhľadovými a navigačnými stromami v práci [7].)
Nech V je množina výhľadov, S je odpovedajúca množina výhľadových slov a B je odpovedajúca množina identifikátorov. Strom T nazveme *výhľadový strom* pre množinu výhľadov V, ak platí:

1. T je stromová reprezentácia množiny reťazcov S,

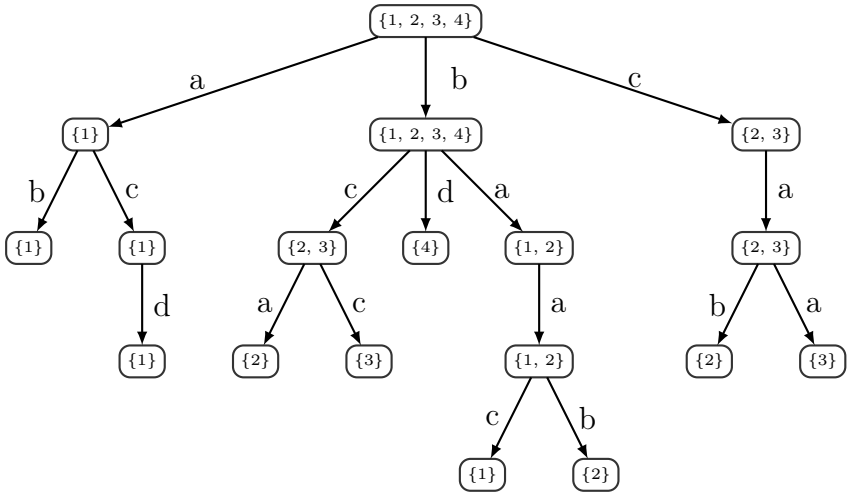
2. každý uzol v_i z T je označený množinou $B_i \subseteq B$ takou, že $b \in B_i$ práve vtedy, ak existuje výhľad $v = (s, b)$ taký, že cesta reprezentujúca výhľadové slovo s obsahuje uzol v_i,

3. každý listový uzol obsahuje jednoprvkovú množinu identifikátorov.

Jednoducho povedané, výhľadový strom pre množinu výhľadov V je stromová reprezentácia množiny príslušných výhľadových slov a každý uzol v strome navyše obsahuje množinu identifikátorov patriacich ku výhľadovým slovám, ktoré sú reprezentované cestami, ktorých je daný uzol súčasťou a každé výhľadové slovo odpovedá práve jednému identifikátoru. Príklad takej reprezentácie je ukázaný na obrázku 4.1.

Definícia 4.14:
Povieme, že výhľadový strom *prijíma reťazec* $w = a_1a_2 \ldots a_m$, ak v ňom existuje cesta $p = (e_1, e_2, \ldots, e_n)$ z koreňa do listu taká, že pre každú hranu e_i platí $l(e_i) = a_i$. O ceste p hovoríme ako o *prijímajúcej ceste pre slovo* w a množinu identifikátorov v listovom uzle cesty p nazveme *výsledok* prijatia slova w.

Poznámka 4.15:
Pretože každý listový uzol výhľadového stromu obsahuje vždy jednoprvkovú množinu identifikátorov, je výsledkom prijatia slova vždy jeden konkrétny identifikátor hrany.



Výhľadový strom pre množinu výhľadov:

$$V = \left\{ \begin{array}{l} (ab,1),(acd,1),(bca,2),(bcc,3),(bd,4), \\ (baac,1),(baab,2),(cab,2),(caa,3) \end{array} \right\}$$

Obr. 4.1: Výhľadový strom pre množinu výhľadov V.

Obrázok 4.1 ilustruje výhľadový strom pre jednoduchú množinu výhľadov. Z tohto stromu môžeme nasledovne jednoducho vygenerovať zdrojový kód rozhodovacej procedúry, ktorá podľa výhľadu (uloženého v poli `la[]`) vyberie správny identifikátor hrany v produkčnom strome, ktorou sa máme ďalej vybrať.

```
1  void decision_procedure()
2  {
3    switch(la[0])
4    {
5    case a:
6      if (la[1] == b || (la[1] == c && la[2] == d))
7      {
8        //procedure following edge #1
9        edge_1();
10       return;
11     }
12   break;
13   case b:
14     switch(la[1])
15     {
16     case a:
17       if (la[2] == a)
18       {
19         if (la[3] == c)
20         {
21           //procedure following edge #1
22           edge_1();
23           return;
24         }
25
26         if (la[3] == b)
27         {
28           //procedure following edge #2
29           edge_2();
```

```
30            return;
31          }
32        }
33      break;
34      case c:
35        if (la[2] == a)
36        {
37          //procedure following edge #2
38          edge_2();
39          return;
40        }
41
42        if (la[2] == c)
43        {
44          //procedure following edge #3
45          edge_3();
46          return;
47        }
48      break;
49      case d:
50        // procedure following edge #4
51        edge_4();
52        return;
53      break;
54      }
55
56    break;
57    case c:
58      if (la[1] == a)
59      {
60        if(la[2] == b)
61        {
62          //procedure following edge #2
63          edge_2();
64          return;
65        }
66        if (la[2] == a)
67        {
68          //procedure following edge #3
69          edge_3();
70          return;
71        }
72      }
73    break;
74    }
75
76    // if no lookahead was matched, throw an exception
77    throw "No lookahead match.";
78  }
```

Listing 4.2: Jedna z možných procedúr implementujúca rozpoznávanie výhľadu podľa výhľadového stromu na obrázku 4.1.

Uvedený zdrojový kód v podstate simuluje prechádzanie príslušným výhľadovým stromom. Postupné zanáranie sa do `case` a `if` vetiev odpovedá jednotlivým cestám v rámci stromu. Uvedený zdrojový kód však nie je optimálny z hľadiska počtu potrebných porovnávaní výhľadu.

Hneď v prvom `case` bloku napríklad zbytočne testujeme výraz `(la[1] == b`

`|| (la[1] == c && la[2] == d)`. Je to preto, že všetky cesty v strome, ktoré začínajú hranou ohodnotenou *a* odkazujú na hranu s identifikátorom *1*.

Tento problém môžeme vyriešiť úpravou stromu tak, že ostránime nadbytočnú časť stromu, pomocou alogoritmu 4.16.

Algoritmus 4.16 (Redukcia výhľadového stromu):
(Prevzaté z [7], strana 78; preložené, upravené.)
Vstup: Výhľadový strom.
Výstup: Výhľadový strom bez nadbytočných podstromov.
Postup:

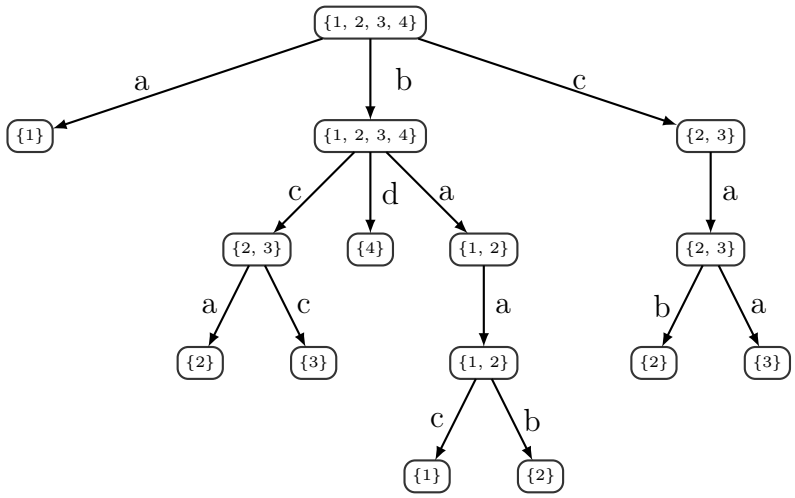
BEGIN

WHILE existuje nelistový uzol n označený jednoprvkovou množinou identifikátorov DO

Zmaž podstrom definovaný uzlom n okrem uzlu n.

END;

Obrázok 4.2 zobrazuje navigačný strom z obrázku 4.1 po aplikácii algoritmu 4.16. Môžeme vidieť, že takmer celý ľavý podstrom bol odstránený.



Obr. 4.2: Navigačný strom po aplikácii algoritmu 4.16.

Algoritmus 4.16 nám teda umožňuje redukovať počet potrebných porovnávaní vo výhľadovom strome v prípadoch, kedy už ostáva na výber len jedna hrana, ktorou je možné pokračovať v produkčnom strome. Touto redukciou však odkladáme porovnanie redukovaného výhľadu a tým prichádzame o možnosť skôr detekovať neodpovedajúci výhľad. To nemusí byť vždy výhodné z hľadiska syntaktickej analýzy, a preto je možnosť redukcie výhľadových stromov ponechaná na užívateľovi.

Samotné výhľadové stromy je možné uchovávať v uzloch odpovedajúcich produkčných stromov, pričom vytvárať výhľadové stromy je nutné len v uzloch produkčných stromov, z ktorých vedie viac ako jedna hrana. Jedinou výnimkou je koreň rekurzívneho stromu. V tomto uzle musí byť výhľadový strom vytvorený vždy, pretože je podľa neho nutné určiť, či sa má pokračovať nejakým rekurzívnym pravidlom.

4.4 Generovanie neterminálnych metód

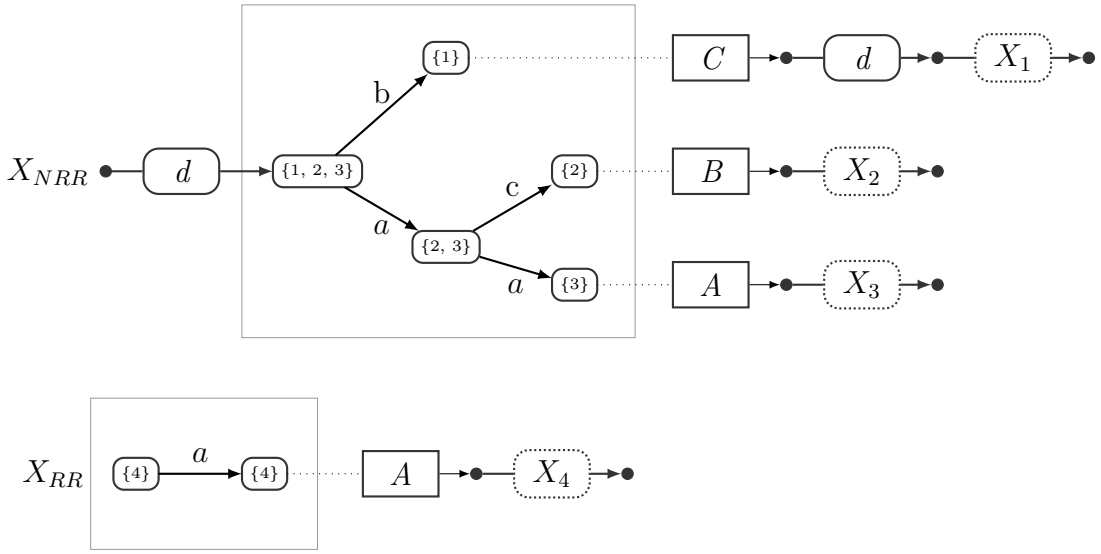
V tejto časti si ukážeme, akú štruktúru bude mať zdrojový kód metód implementujúcich rozvoj neterminálov prívetivej gramatiky, ktoré predstavujú jadro vygenerovaného staticky rekurzívne zostupného analyzátora. Na ich vygenerovanie použijeme doposiaľ predstavené dátové štruktúry a budeme sa zaoberať aj problémom formátovania tohto kódu.

Každá z týchto metód bude odpovedať práve jednému neterminálu a tie sa budú podľa potreby rekurzívne volať. V rámci týchto metód bude simulovaný prechod odpovedajúcimi produkčnými stromami daného neterminálu a pri vyberaní vetvy, ktorou sa v týchto stromoch bude pokračovať bude využitý výhľadový strom v odpovedajúcom uzle produkčného stromu.

Pre ilustráciu uvažujme jednoduché množiny terminálov, neterminálov a pravidiel:

$$T = \{a,\ b,\ c,\ d\}$$
$$N = \{X,\ A,\ B,\ C\}$$
$$P = \left\{ \begin{array}{l} X \to dA,\ X \to dB,\ X \to dCd, X \to XA \\ A \to aa,\ B \to ac,\ C \to bc \end{array} \right\}$$

Podľa týchto množín môžeme vytvoriť produkčné stromy pre neterminál X. Tieto stromy sú zobrazené na obrázku 4.3. Uzol označený X_{NRR} predstavuje koreň produkčného stromu obsahujúceho nerekurzívne pravidlá neterminálu X. Uzol označený X_{RR} predstavuje koreň produkčného stromu obsahujúceho rekurzívne pravidlá neterminálu X. Obdĺžnikom je označený rozšírený uzol stromu obsahujúci výhľadový strom, podľa ktorého je možné rozhodnúť, akou vetvou sa máme vybrať. Bodkované hrany slúžia ako referencie príslušných identifikátorov.



Obr. 4.3: Produkčné stromy pre neterminál X.

Týmto produkčným stromom bude odpovedať nasledovná metóda:

```
1  void nont_X()
2  {
3    // non-recursive tree
4    matchTerminal(Terminal::T_D);
5
6    if (compareView(0, Terminal::T_A))
```

```
7    {
8      if (compareView(1, Terminal::T_A))
9      {
10       nont_A();
11     }
12     else if (compareView(1, Terminal::T_C))
13     {
14       nont_B();
15     }
16     else
17     {
18       throw syntax_error("Syntax error");
19     }
20   }
21   else if (compareView(0, Terminal::T_B))
22   {
23     nont_C();
24     matchTerminal(Terminal::T_D);
25   }
26   else
27   {
28     throw syntax_error("Syntax error");
29   }
30
31   // recursive tree
32   while(true)
33   {
34     if (compareView(0, Terminal::T_A))
35     {
36       nont_A();
37     }
38     else
39     {
40       break;
41     }
42   }
43
44   // NLRF
45   if (compareView(0, Terminal::S_EOI))
46   {
47     return;
48   }
49   else
50   {
51     throw syntax_error("Syntax error");
52   }
53 }
```

Túto metódu je možné rozdeliť na tri časti. Prvú časť reprezentujú riadky 3 – 29 a odpovedá prechodu nerekurzívnym produkčným stromom neterminálu X. Môžeme si všimnúť, že táto časť začína metódou `matchTerminal`, ktorej je ako parameter predaný identifikátor terminálu d. Keďže z koreňa nerekurzívneho stromu vedie len jedna hrana (reprezentujúca rozvoj spomínaného terminálu d), nie je nutné pred prechodom touto hranou kontrolovať výhľad. V prípade, že by na vstupe nenasledoval terminál d volanie metódy `matchTerminal` zlyhá a bude vyvolaná výnimka, ktorá syntaktickú analýzu preruší.

Po prechode hranou reprezentujúcou terminál d môže syntaktický analyzátor

pokračovať troma hranami (reprezentujúcimi neterminály A, B alebo C). V tomto prípade je teda podľa odpovedajúceho výhľadového stromu vygenerovaná rozhodovacia časť, ktorá na základe výhľadu vyberie správnu hranu. K porovnávaniu s aktuálnym výhľadom slúži metóda `compareView`, ktorá ako parametre dostáva pozíciu vo výhľade, ktorá má byť porovnaná s predaným identifikátorom terminálu. V prípade zhody vracia táto metóda hodnotu `true`, inak vracia hodnotu `false`. V prípade, že výhľad odpovedá nejakej očakávanej kombinácii, je zavolaná metóda odpovedajúca rozvoju patričného neterminálu, inak program skončí vo vetve obsahujúcej vyvolanie výnimky `syntax_error` a syntaktická analýza bude prerušená.

Druhá časť tejto metódy predstavuje spracovanie rekurzívneho produkčného stromu neterminálu X a je reprezentovaná riadkami 31 – 42. Vygenerovanie tejto časti je závislé na tom, či neterminál má nejaké rekurzívne pravidlo. V prípade, že nemá, je táto časť úplne vynechaná.

Časť metódy spracovávajúca rekurzívny produkčný strom sa líši od časti pre nerekurzívny strom v niekoľkých ohľadoch. V prvom rade, musí byť v koreni rekurzívneho produkčného stromu vždy vytvorený výhľadový strom aj v prípade, že z neho vedie len jedna hrana. Je to preto, aby sme sa dokázali na základe výhľadu rozhodnúť, či sa vôbec má rekurzívnym stromom prechádzať. Druhou odlišnosťou je, že spracovanie rekurzívneho stromu je uzavreté v nekonečnom cykle, čo umožňuje jednoducho simulovať viacnásobné rekurzívne zanorenie. Tento cyklus je prerušený (riadok 40) v prípade, že pri vstupe do rekurzívneho stromu neodpovedá výhľad žiadnemu výhľadu z výhľadového stromu v koreni rekurzívneho stromu. Inak prebieha spracovanie rovnako ako v časti pre nerekurzívny strom.

Posledná časť metódy (riadky 44 – 52) slúži ku kontrole, či po spracovaní nerekurzívneho a rekurzívneho stromu neterminálu X odpovedajú terminály na vstupe nejakému výhľadu neterminálu X mimo ľavej rekurzie (*NLRF(X)*). Táto kontrola umožňuje včas odhaliť neočakávaný výhľad po ukončení rozvoja neterminálu X. Okrem tejto kontroly ďalej plní dôležitú úlohu v metóde implementujúcej rozvoj počiatočného neterminálu prívetivej gramatiky, kde efektívne kontroluje, či bol spracovaný celý vstup (vo výhľade je očakávaný terminál označujúci koniec vstupu).

Metódy implementujúce rozvoj neterminálov predstavujú najpremenlivejšiu a najkomplikovanejšiu časť zdrojových kódov vygenerovaných analyzátorov a translátorov. Zložitosť produkčných a výhľadových stromov rastie so zložitosťou gramatiky a požadovanou dĺžkou výhľadu, čo priamo zvyšuje zložitosť vygenerovaného kódu týchto metód. Bolo preto nutné zvoliť formát implementácie, ktorý by zabezpečil užívateľovi čo najlepšiu možnú orientáciu a najjednoduchšie pochopenie tohto vygenerovaného kódu.

Jednou z možností bolo dekomponovať tieto metódy na menšie celky, napríklad samostatné metódy implementujúce prechod rekurzívnym a nerekurzívnym stromom daného neterminálu a kontroly výhľadu mimo ľavej rekurzie. Problém je, že aj tieto celky môžu byť stále pomerne veľké, a preto by bolo potrebné ich ďalej dekomponovať, napríklad rozdelením na metódy implementujúce prechody časťami produkčných stromov, prípadne v extrémnych prípadoch iba jednotlivých uzlov. Týmto by vznikol kód, v ktorom by síce jednotlivé metódy mohli mať pomerne prijateľnú veľkosť, avšak celková implementácia rozvoja neterminálu by už obsahovala príliš veľký počet týchto metód. Zároveň by bolo nutné tieto nejak

identifikovať (pomenovať) podľa toho, čo implementujú, čím by vznikali neprirodzené názvy, napríklad `nont_X_recursive_tree_node_42`, čo by znamenalo pravý opak zrozumiteľnosti.

Preto sme sa rozhodli implementovať rozvoj každého neterminálu jednou metódou. K zrozumiteľnosti tejto metódy bude prispievať vhodné formátovanie použitých konštruktov, zjednodušovanie zápisu podmienok pri kontrole výhľadov a intuitívne pomenovanie použitých konštánt, identifikátorov a metód.

Mená jednotlivých symbolov gramatiky sú zvolené tak, aby bolo jednoduché tieto symboly v rámci vygenerovaného kódu rýchlo identifikovať. To je dosiahnuté pridaním vhodného prefixu k pomenovaniu symbolu. Napríklad metódy implementujúce rozvoj neterminálu majú vždy prefix `nont_` a identifikátory terminálov vždy prefix `T_`. Podobne mená použitých metód sú volené tak, aby bol na prvý pohľad jasný ich účel. Napríklad `compareView` na porovnávanie výhľadu a `matchTerminal` na porovnávanie aktuálneho terminálu na vstupe.

Pri kontrole výhľadu je používaný výhradne konštrukt `if .. else if .. else`, ktorý poskytuje väčšiu flexibilitu než konštrukt `switch .. case`. Pri generovaní kódu jednotlivých výhľadových stromov tiež dochádza k zlučovaniu podmienok tak, aby sa zjednodušil ich zápis a zvýšila prehľadnosť za zachovania významu. Napríklad nasledovnú podmienku:

```
1  if (compareView(0, Terminal::T_A))
2  {
3    if (compareView(1, Terminal::T_B))
4    {
5      if (compareView(2, Terminal::T_A))
6      {
7        nont_A();
8      }
9    }
10 }
11 else
12 {
13   ...
14 }
```

je možné zjednodušene zapísať ako:

```
1  if (compareView(0, Terminal::T_A)
2     && compareView(1, Terminal::T_B)
3     && compareView(2, Terminal::T_A))
4  {
5    nont_A();
6  }
7  else
8  {
9    ...
10 }
```

čím sa efektívne zníži zanorenie jednotlivých blokov zdrojového kódu a zároveň zvýši čitateľnosť danej podmienky.

Podobne je možné zjednodušiť nasledujúci tvar podmienky:

```
1  if (compareView(0, Terminal::T_A))
2  {
3    nont_A();
4  }
```

```
5  else if (compareView(0, Terminal::T_B))
6  {
7    nont_A();
8  }
9  else if (compareView(0, Terminal::T_C))
10 {
11   nont_A();
12 }
13 else
14 {
15   ...
16 }
```

na tvar:

```
1  if (compareView(0, Terminal::T_A)
2     || compareView(0, Terminal::T_B)
3     || compareView(0, Terminal::T_C))
4  {
5    nont_A();
6  }
7  else
8  {
9    ...
10 }
```

čím sa efektívne zníži dĺžka vygenerovaného zdrojového kódu a znova zvýši čitateľnosť danej podmienky.

Na druhej strane sa vyvarujeme vytváraniu podmienok, ktoré obsahujú zároveň operátor `&&` aj `||`, pretože by pri väčšom množstve porovnaní v jednej podmienke mohli naopak čitateľnosť znižovať.

Kapitola 5

CppKind

V tejto kapitole popíšeme implementovanú aplikáciu CppKind z užívateľského pohľadu. Budeme sa venovať nastaveniu a použitiu samotnej aplikácie. CppKind je implementácia generátora prívetivých syntaktických analyzátorov a translátorov v jazyku *C++* na základe popisu jazyka v konfiguračnom súbore.

5.1 Štruktúra projektu a kompilácia

Odovzdaná práca má nasledujúcu adresárovú štruktúru:

<*root_dir*>
- docs
  - thesis ..... Dokument diplomovej práce
  - documentation ..... Programová dokumentácia aplikácie CppKind
- examples ..... Ukážky konfiguračných súborov a vstupu
- sources ..... Zdrojové kódy aplikácie CppKind

Kompilácia zdrojových kódov

Okrem zdrojových kódov aplikácie CppKind obsahuje adresár `.\sources` ešte dva pomocné súbory *CMakeLists.txt* a *sources.cmake*. Tieto súbory sú určené pre použitie s programom *CMake* [16], ktorý umožňuje vytváranie projektov pre rôzne platformy. Pre použitie s našim projektom je vyžadované mať nainštalovaný *CMake* verzie aspoň 3.0.0.

Súbor *sources.cmake* obsahuje zoznam zdrojových súborov aplikácie CppKind a je využívaný pri spracovaní súboru *CMakeLists.txt*.

Súbor *CMakeLists.txt* definuje projekty a ich nastavenia, ktoré majú byť z dodaných zdrojových kódov vygenerované. Týmito projektmi sú:

1. statická knižnica *CppKindLib*, ktorá obsahuje celú funkcionalitu aplikácie CppKind,

2. spustiteľná aplikácia CppKind, využívajúca knižnicu *CppKindLib*.

Ďalej si z nastavení môžeme všimnúť, že tieto projekty vyžadujú kompilátor podporujúci štandard *C++11*. V nasledujúcej časti si popíšeme ako vytvoriť a skompilovať projekt s použitím kompilátorov *Microsoft Visual Studio* a *g++*.

Tieto návody predpokladajú, že užívateľ má nainštalované a správne nastavené všetky prerekvizity potrebné ku kompilácii (*CMake*, *Microsoft Visual Studio* / *g++*). V prípade problémov je vhodné konzultovať manuály jednotlivých programov a programu *CMake* [17].

Kompilácia s Microsoft Visual Studio

Pre kompiláciu aplikácie CppKind pomocou programu *Microsoft Visual Studio* je možné postupovať nasledujúcimi krokmi:

1. Vytvoriť adresár, kde bude vytvorené riešenie (solution) pre program *Microsoft Visual Studio*.
2. V tomto adresári je potom potrebné spustiť program *CMake*, ktorému je ako parameter predaný adresár obsahujúci súbor *CMakeLists.txt*. Tento krok vygeneruje požadované riešenie pre program *Microsoft Visual Studio*.
3. Potom je vo *Visual Studiu* možné skompilovať projekt CppKind, čím získame spustiteľnú aplikáciu aj statickú knižnicu *CppKindLib*.

Kompilácia pre operačný systém Windows 10 bola testovaná použitím programu *Microsoft Visual Studio 2015 Community* (voľne dostupný pre osobné a vzdelávacie účely).

Kompilácia pomocou g++

Kompilácia pomocou programu *g++* prebieha podobne ako v predchádzajúcej časti. Najskôr je nutné vytvoriť adresár, kde budú zdrojové kódy kompilované. V tomto adresári je potrebné spustiť aplikáciu *CMake*, ktorej je ako parameter predaný adresár obsahujúci súbor *CMakeLists.txt*, čím sa vytvorí makefile súbor použiteľný kompilátorom *g++*. Toto všetko je možné vykonať napríklad zadaním nasledujúcich príkazov na príkazovom riadku:

```
mkdir build
cd build
cmake <path_to_cmakelists_directory>
make
```

`<path_to_cmakelists_directory>` je adresár obsahujúci odpovedajúci súbor *CMakeLists.txt*. Spustenie príkazu `make` vykoná inštrukcie zapísané vo vygenerovanom makefile súbore, čím vznikne spustiteľná aplikácia CppKind spolu so statickou knižnicou *libCppKindLib*.

Kompilácia pre operačný systém Linux (Ubuntu 15.04) bola testovaná použitím programu *g++* verzie 4.9.2.

5.2 Konfiguračný súbor

Konfiguračný súbor obsahuje všetky potrebné nastavenia pre správny beh programu CppKind.

Príklad 5.1 predstavuje ukážku konfiguračného súboru pre gramatiku jednoduchého aritmetického výrazu.

Príklad 5.1 (Konfiguračný súbor):

```
[Language]
    ParserClassName = AriExp
    namespace = Test::Analyzators
    lookahead = 1
    startsymbol = S
    numterminal = num
    ignorewhitespace = yes
    utilfile = yes
    casesensitive = no
    viewmode = depthminimal
    minimalworkingexample = yes
    withtrace = yes
    singlecomment = //
    commentstart = /*
    commentend = */
[Symbols]
    plus = +
    star = *
    lpar = (
    rpar = )
    minus = -
[OutTerminals]
    OutPlus = +
    OutMinus = -
    OutStar = *
    OutEqual = =
[Nonterminals]
    S
    E
    T
    F
[Actions]
a_getnum; private; void; getnum;;
std::string atr = attributes_[attributes_.size() - 1].lexeme();
stack_.push_back(std::stoi(atr));
out() << stack_[stack_.size() - 1];
<!end>

a_add; private; void; add;;
stack_[stack_.size() - 2] += stack_[stack_.size() - 1];
stack_.pop_back();
<!end>

a_sub; private; void; sub;;
stack_[stack_.size() - 2] -= stack_[stack_.size() - 1];
stack_.pop_back();
<!end>
```

```
a_multi; private; void; multi;;
stack_[stack_.size() - 2] *= stack_[stack_.size() - 1];
stack_.pop_back();
<!end>

a_print; private; void; print;;
out() << stack_[0];
<!end>

a_space; private; void; space;;
out() << " ";
<!end>

[Action-members]
std::vector<int> stack_;
[Productions]
    [S]::=[E]<OutEqual>{a_space}{a_print}
    [E]::=[T]
    [E]::=[E](plus)[T]{a_add}<OutPlus>{a_space}
    [E]::=[E](minus)[T]{a_sub}<OutMinus>{a_space}
    [T]::=[F]
    [T]::=[T](star)[F]{a_multi}<OutStar>{a_space}
    [F]::=(num){a_getnum}{a_space}
    [F]::=(lpar)[E](rpar)
```

Sekcie

Štruktúra konfiguračného súboru je rozdelená na jednotlivé sekcie, ktoré predstavujú rôzne časti definície vstupnej gramatiky a príznakov pre generátor CppKind. Poradie sekcií nie je nutné presne dodržiavať, avšak je nutné dodržať správne poradie deklarácií. To znamená, že napríklad neterminál nemôže byť použitý v deklarácii pravidla, kým nebol sám deklarovaný. Toto obmedzenie sa týka neterminálov, terminálov, výstupných terminálov a sémantických akcií.

Každá sekcia začína svojou hlavičkou, ktorú predstavuje jej identifikátor uzavretý v hranatých zátvorkách. Na veľkosti jednotlivých znakov v identifikátore nezáleží, teda:

`[Nonterminals]`

a

`[NoNtErMiNaLs]`

definujú rovnakú hlavičku sekcie neterminálov. Rovnako nezáleží na bielych znakoch pred a po hranatých zátvorkách, ktoré sú ignorované, avšak biele znaky medzi hranatými zátvorkami sú neprípustné.

Tabuľka 5.1 vymenúva hlavičky prípustné v konfiguračnom súbore pre program CppKind.

Prázdne riadky a riadky začínajúce znakmi `//` (komentáre) sú ignorované. Neplatí to však vo všetkých sekciách. Ak je riadok súčasťou nejakého zdrojového kódu (napríklad v definícii sémantických akcií), tak daný riadok nie je ignorovaný.

Hlavička	Popis
Language	Nastavenia spojené s niektorými aspektmi gramatiky a príznakov pre generátor CppKind.
Symbols	Deklarácie a definície symbolov.
Keywords	Deklarácie a definície kľúčových slov.
Outterminals	Deklarácie a definície výstupných terminálov.
Nonterminals	Deklarácie neterminálov.
Actions	Deklarácie a definície sémantických akcií.
Action-includes	Hlavičkové súbory potrebné k správnemu preloženiu a behu sémantických akcií.
Action-init	Inicializačný kód potrebný k správnemu nastaveniu sémantických akcií.
Action-cleanup	Kód zodpovedný za správne uvoľnenie prostriedkov sémantických akcií.
Action-members	Deklarácia premenných, ktoré sú používané akciami.
Productions	Pravidlá (produkcie) gramatiky.

Tabuľka 5.1: Zoznam prípustných hlavičiek konfiguračného súboru pre generátor CppKind.

Sekcia Language

Sekcia *Language* obsahuje nastavenia aplikácie, definície špeciálnych symbolov a príznakov pre generátor CppKind. Jednotlivé *záznamy* v tejto sekcii sú uvádzané na samostatných riadkoch a majú tvar:

$$kľúč = hodnota,$$

pričom *kľúč* je reťazec identifikujúci určité nastavenie a *hodnota* závisí na danom kľúči. Na veľkosti znakov v kľúči nezáleží, avšak veľkosť znakov v hodnote je braná do úvahy. Hodnoty v záznamoch sú vždy jedným z nasledujúcich štyroch typov:

1. *boolean* - typ, ktorý môže obsahovať hodnoty typu *pravda (true)* alebo *nepravda (false)*. Pri spracovávaní záznamov predstavujú reťazce *y*, *yes* a *true* hodnotu true a akýkoľvek iný reťazec predstavuje hodnotu false.

2. *int* - typ, ktorý predstavuje číselnú hodnotu. Dodatočné podmienky, ktoré sú kladené na hodnotu reprezentovanú týmto typom pre jednotlivé záznamy, uvádzame pri popise konkrétneho záznamu.

3. *char* - typ, ktorý predstavuje hodnotu obsahujúcu jeden znak. Dodatočné podmienky, ktoré sú kladené na hodnotu reprezentovanú týmto typom pre jednotlivé záznamy, uvádzame pri popise konkrétneho záznamu.

4. *string* - typ, ktorý predstavuje reťazec znakov. Dodatočné podmienky, ktoré sú kladené na hodnotu reprezentovanú týmto typom pre jednotlivé záznamy, uvádzame pri popise konkrétneho záznamu.

5. *enum* - typ, ktorý predstavuje preddefinovanú množinu hodnôt. Prípustné hodnoty tohto typu pre konkrétny záznam sú vždy popísané s konkrétnym záznamom.

Všetky možné kľúče aj s ich možnými hodnotami sú uvedené nižšie.

parserclassname = string

parserclassname očakáva hodnotu typu reťazec, táto hodnota bude použitá ako meno triedy vygenerovaného parseru. Prípustná hodnota začína písmenom a nasledujúce znaky môžu byť písmeno alebo číslo. Záznam nie je povinný, ak nie je uvedený, bude ako meno vygenerovanej triedy použitá hodnota *Parser*.

namespace = string

namespace očakáva hodnotu typu reťazec, ktorá reprezentuje menný priestor, v ktorom má byť vygenerovaná trieda parseru. Očakávaná hodnota začína písmenom a pokračuje písmenom alebo číslom. V rámci záznamu je možné zadať viac menných priestorov pomocou oddeľovača "::", prázdne reťazce medzi dvoma oddeľovačmi sú ignorované, teda záznam tvaru:

$$namespace = n1::n2::::n3::::n4,$$

odpovedá mennému priestoru *n1::n2::n3::n4*. Záznam nie je povinný, ak nie je uvedený, bude trieda parseru vygenerovaná bez zasadenia do menného priestoru.

casesensitive = boolean

casesensitive očakáva hodnotu typu boolean, ktorá definuje, či má lexikálny analyzátor daného parseru rozlišovať medzi veľkosťou znakov v analyzovanom texte. Záznam nie je povinný a prednastavená hodnota je false.

utilfile = boolean

util očakáva hodnotu typu boolean, ktorá definuje, či má byť vygenerovaný súbor s utilitami (viď sekciu 6.1) pre daný parser. Záznam nie je povinný a prednastavená hodnota je true.

chardelim = char

chardelim očakáva hodnotu typu char, ktorá definuje znak použitý lexikálnym analyzátorom na rozpoznanie hodnoty typu znak v analyzovanom texte. Záznam nie je povinný a prednastavená hodnota sa môže líšiť v závislosti na zázname *stringdelim* - pokiaľ nie je hodnota záznamu *stringdelim* nastavená na ', je tento znak použitý ako prednastavená hodnota, inak je použitá hodnota ".

stringdelim = char

stringdelim očakáva hodnotu typu char, ktorá definuje znak použitý lexikálnym analyzátorom na rozpoznanie hodnoty typu string v analyzovanom texte. Záznam nie je povinný a prednastavená hodnota sa môže líšiť v závislosti na zázname *chardelim* - pokiaľ nie je hodnota záznamu *chardelim* nastavená na ", je tento znak použitý ako prednastavená hodnota, inak je použitá hodnota '.

lookahead = int

lookahead očakáva hodnotu typu int, ktorá predstavuje maximálnu možnú dĺžku výhľadu použitého pri syntaktickej analýze vstupného textu. Očakáva sa hodnota väčšia ako 0. Položka nie je povinná a má prednastavenú hodnotu 1.

V aktuálnej verzii generátora je maximálna hodnota tejto položky obmedzená na 5.

viewmode = enum

viewmode očakáva hodnotu typu enum, ktorá indikuje spôsob použitia výhľadu pri syntaktickej analýze vstupného textu. Množina prípustných hodnôt je *{full, depthminimal}*. Pri použití hodnoty *full* je vždy použitá plná dĺžka výhľadu; pri použití hodnoty *depthminimal* je použitá minimálna nutná dĺžka výhľadu (viď algoritmus 4.16). Záznam nie je povinný a má prednastavenú hodnotu *full*.

numterminal = string

numterminal očakáva hodnotu typu string, ktorá reprezentuje meno terminálu predstavujúceho celé kladné číslo, ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Táto položka nie je povinná a nemá žiadnu prednastavenú hodnotu.

realterminal = string

realterminal očakáva hodnotu typu string, ktorá reprezentuje meno terminálu predstavujúceho kladné desatinné číslo, ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Táto položka nie je povinná a nemá žiadnu prednastavenú hodnotu.

allowenotation = boolean

allowenotation očakáva hodnotu typu boolean, ktorá určuje či bude lexikálny analyzátor prijímať aj desatinné čísla v tzv. vedeckej notácii (napríklad *7.56E-23*). Táto položka je závislá na položke *realterminal* a môže mať kladnú hodnotu len v prípade, že je položka *realterminal* definovaná. Položka nie je povinná a má prednastavenú hodnotu *false*.

numericsuffixes = string

numericsuffixes očakáva hodnotu typu reťazec, ktorá definuje množinu prípustných sufixov číselných konštánt.

V niektorých prípadoch je výhodné mať možnosť spolu s číselnou konštantou bližšie špecifikovať aj jej typ. Príkladom môže byť zápis číselných konštánt v jazyku C#, ktorý definuje tzv. numerické sufixy. Napríklad zápis *123.4f* značí, že číselná konštanta *123.4* by mala byť reprezentovaná typom *float* jazyka C#.

V snahe poskytnúť užívateľovi možnosť podobného zápisu číselných konštánt, umožňuje táto položka definovať množinu znakových reťazcov, ktoré môžu nasledovať priamo za číselnou zložkou a budú považované za súčasť číselnej konštanty. Prípustné sú iba neprázdne znakové reťazce obsahujúce len písmená. Jednotlivé reťazce sú v položke navzájom oddelené znakom `';'` a prípadný zápis môže vyzerať nasledovne:

$$numericsuffixes = f;F;d;D;UL;ul$$

V prípade, že je použitý lexikálny analyzátor, ktorý nerozlišuje medzi malými a veľkými variantami písmen, sú vo vygenerovanom kóde všetky hodnoty prevedené na ich malú variantu a duplicitné hodnoty sú ignorované.

charterminal = string

charterminal očakáva hodnotu typu string, ktorá reprezentuje meno terminálu predstavujúceho znakovú konštantu (literál), ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Tento terminál nie je povinný a nemá žiadnu prednastavenú hodnotu. Pri nezadaní tohto záznamu nebude lexikálny

analyzátor vo vstupnom texte rozlišovať znakové konštanty vôbec. Pri rozpoznávaní tejto konštanty lexikálny analyzátor používa hodnotu definovanú v zázname *chardelim* alebo jeho prednastavenú hodnotu.

stringterminal = string

stringterminal očakáva hodnotu typu string, ktorá reprezentuje meno terminálu predstavujúceho reťazcovú konštantu (literál), ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Tento terminál nie je povinný a nemá žiadnu prednastavenú hodnotu. Pri nezadaní tohto záznamu nebude lexikálny analyzátor vo vstupnom texte rozlišovať reťazcové konštanty vôbec. Pri rozpoznávaní tejto konštanty lexikálny analyzátor používa hodnotu definovanú v zázname *stringdelim* alebo jeho prednastavenú hodnotu.

identifierterminal = string

identifierterminal očakáva hodnotu typu string, ktorá reprezentuje meno terminálu predstavujúceho identifikátor, ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Tento terminál nie je povinný a nemá žiadnu prednastavenú hodnotu. Pri nezadaní tohto záznamu nebude lexikálny analyzátor vo vstupnom texte rozlišovať identifikátory vôbec.

Záznamy *numterminal*, *realterminal*, *charterminal*, *stringterminal*, *identifierterminal* spolu v rámci danej gramatiky definujú množinu *špeciálnych terminálov*, ktoré nemajú konkrétny tvar a popisujú množiny hodnôt.

ignorewhitespace = boolean

ignorewhitespace očakáva hodnotu typu boolean, ktorá špecifikuje, či má lexikálny analyzátor automaticky preskakovať biele znaky (medzery, tabulátory a nové riadky) vo vstupnom texte. Táto položka nie je povinná a má prednastavenú hodnotu *true*.

V prípade, že má táto položka hodnotu *false*, musia byť zároveň definované identifikátory terminálov pre nový riadok, tabulátor a medzeru (viď nižšie).

spaceterminal = string

spaceterminal očakáva hodnotu typu reťazec, ktorá reprezentuje meno terminálu predstavujúceho medzeru (`' '`), ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Tento terminál nie je povinný a nemá žiadnu prednastavenú hodnotu.

tabulatorterminal = string

tabulatorterminal očakáva hodnotu typu reťazec, ktorá reprezentuje meno terminálu predstavujúceho tabulátor (`'\t'`), ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Tento terminál nie je povinný a nemá žiadnu prednastavenú hodnotu.

newlineterminal = string

newlineterminal očakáva hodnotu typu reťazec, ktorá reprezentuje meno terminálu predstavujúceho nový riadok (`'\n'`), ktoré je použité lexikálnym analyzátorom pri spracovaní vstupného textu. Tento terminál nie je povinný a nemá žiadnu prednastavenú hodnotu.

Poznámka 5.2:
Položky *spaceterminal*, *tabulatorterminal* a *newlineterminal* sú závislé na nasta-

vení položky *ignorewhitespace*. V prípade, že má položka *ignorewhitespace* definovanú hodnotu *false*, musia byť definované aj tieto položky, zatiaľ čo v opačnom prípade nie sú hodnoty týchto položiek vôbec brané do úvahy.

singlecomment = string

singlecomment očakáva hodnotu typu reťazec, ktorá definuje oddeľovač jednoriadkových komentárov vo vstupnom súbore. Prípustnou hodnotou je neprázdny znakový reťazec, ktorý neobsahuje alfanumerické a biele znaky. Táto položka nie je povinná a nemá žiadnu prednastavenú hodnotu.

commentstart = string

commentstart očakáva hodnotu typu reťazec, ktorá definuje počiatočný oddeľovač viacriadkových komentárov vo vstupnom súbore. Prípustnou hodnotou je neprázdny znakový reťazec, ktorý neobsahuje alfanumerické a biele znaky. Táto položka nie je povinná a nemá žiadnu prednastavenú hodnotu, avšak musí byť definovaná spoločne s položkou *commentend*.

commentend = string

commentend očakáva hodnotu typu reťazec, ktorá definuje koncový oddeľovač viacriadkových komentárov vo vstupnom súbore. Prípustnou hodnotou je neprázdny znakový reťazec, ktorý neobsahuje alfanumerické a biele znaky. Táto položka nie je povinná a nemá žiadnu prednastavenú hodnotu, avšak musí byť definovaná spoločne s položkou *commentstart*.

extended = boolean

extended očakáva hodnotu typu boolean, ktorá určuje, či sa má použiť rozšírená prívetivá gramatika, čo znamená oslabenie kontroly konfliktov vypočítaných výhľadov. V prípade, že je nejaké pravidlo prefixom iného pravidla a je možné použiť obe pravidlá, potom je použité dlhšie pravidlo. Táto položka nie je povinná a má prednastavenú hodnotu *false*.

withtrace = boolean

withtrace očakáva hodnotu typu boolean, ktorá indikuje, či má vygenerovaný analyzátor obsahovať nástroje na jednoduché zaznamenávanie krokov výpočtu. Táto položka nie je povinná a má prednastavenú hodnotu *false*.

startsymbol = string

startsymbol očakáva hodnotu typu string, ktorá reprezentuje meno počiatočného neterminálu definovanej gramatiky. Toto meno musí byť zároveň deklarované v sekcii *Nonterminals*. Tento záznam je povinný.

minimalworkingexample = boolean

minimalworkingexample očakáva hodnotu typu boolean, ktorá signalizuje, či má byť vygenerovaný súbor s ukážkou základného použitia vygenerovaného analyzátora. Tento záznam nie je povinný a má prednastavenú hodnotu *false*.

Sekcia Symbols

Sekcia *Symbols* slúži k definícii symbolického terminálu (ďalej len symbol). Za symbol je v rámci gramatiky považovaný terminál s preddefinovaným tvarom, ktorý

nie je číslo, identifikátor, znaková či reťazcová konštanta alebo definované kľúčové slovo. Každá definícia symbolu je v rámci sekcie uvádzaná na samostatnom riadku a má nasledujúci tvar:

$$meno = tvar,$$

kde *meno* slúži na identifikáciu terminálu v rámci gramatiky a zároveň je súčasťou identifikátora terminálu v rámci kódu vygenerovaného analyzátora. Ako meno sa predpokladá reťazec, ktorý začína písmenom a pokračuje písmenom, číslicou alebo znakom `'_'`. *Meno* symbolu musí byť v rámci terminálov gramatiky jedinečné.

Tvar symbolu je reťazec, ktorý je lexikálnym analyzátorom používaný pri analýze vstupného textu - vstupný text je porovnávaný s touto hodnotou a pri zhode je prijatý daný symbol. Symbol môže mať tvar akéhokoľvek neprázdneho znakového reťazca bez bielych znakov. *Tvar* symbolu musí byť jedinečný v rámci tvarov všetkých terminálov danej gramatiky.

Sekcia Keywords

Sekcia *Keywords* slúži k definícii terminálu kľúčového slova (ďalej len kľúčové slovo). Za kľúčové slovo je v rámci gramatiky považovaný terminál s preddefinovaným tvarom, ktorý nie je číslo, identifikátor, znaková či reťazcová konštanta alebo definovaný symbol. Každá definícia kľúčového slova je v rámci sekcie uvádzaná na samostatnom riadku a má nasledujúci tvar:

$$meno = tvar,$$

kde *meno* slúži na identifikáciu terminálu v rámci gramatiky a zároveň je súčasťou identifikátora terminálu v rámci kódu vygenerovaného analyzátora. Ako *meno* sa predpokladá reťazec, ktorý začína písmenom a pokračuje písmenom, číslicou alebo znakom `'_'`. *Meno* symbolu musí byť v rámci terminálov gramatiky jedinečné.

Tvar kľúčového slova je reťazec, ktorý je lexikálnym analyzátorom používaný pri analýze vstupného textu - vstupný text je porovnávaný s touto hodnotou a pri zhode je prijatý ako dané kľúčové slovo. Kľúčové slovo môže mať tvar akéhokoľvek neprázdneho znakového reťazca, ktorý začína písmenom alebo znakom `'_'` a pokračuje písmenom, číslicou alebo znakom `'_'`. *Tvar* kľúčového slova musí byť jedinečný v rámci tvarov všetkých terminálov danej gramatiky.

Symboly, kľúčové slová a špeciálne terminály spolu tvoria množinu všetkých *terminálov* danej gramatiky[1].

Sekcia Outterminals

V sekcii *Outterminals* sa nachádzajú definície výstupných terminálov. Podobne ako pri symboloch a kľúčových slovách má záznam v tejto sekcii tvar:

$$meno = tvar,$$

[1]Terminál pre koniec vstupu je použitý len v internej implementácii a užívateľ by ho nemal mať potrebu deklarovať.

kde *meno* je identifikátor výstupného terminálu v rámci gramatiky a zároveň je súčasťou deklarácie metódy príslušného neterminálu v kóde vygenerovaného translátora. Ako *meno* sa predpokladá reťazec, ktorý začína písmenom a pokračuje písmenom, číslicou alebo znakom `'_'`. Meno výstupného terminálu musí byť jedinečné v rámci množiny výstupných terminálov, avšak môže byť zhodné s menom nejakého iného terminálu (kľúčového slova, symbolu alebo špeciálneho terminálu). Veľkosť znakov v mene výstupného terminálu nie je podstatná, pretože generátor prevedie všetky znaky v mene na ich malú variantu.

Tvar výstupného terminálu môže byť ľubovoľný reťazec. Tento reťazec je pri použití výstupného terminálu v translátore zapísaný na definovaný výstup. Tvar výstupného terminálu je pri spracovaní zbavený bielych znakov na začiatku a konci reťazca, ktorý ho definuje. Ak je potrebné vytvoriť výstupný terminál, ktorého tvar obsahuje na niektorej strane biele znaky, je možné uzavrieť tvar medzi znaky `""`. Táto dvojica znakov je následne pri spracovaní záznamu z tvaru výstupného terminálu odstránená. Tabuľka 5.2 ukazuje použitie znakov `""` v definícii výstupného terminálu spolu s výstupným reťazcom, ktorý daná definícia predstavuje.

Definícia	Výstup
`outterm_1 = "hello world"`	`hello world`
`outterm_2 = "  hello world  "`	`␣␣␣hello world␣␣␣`
`outterm_3 = "  "hello world"  "`	`␣␣␣"hello world"␣␣␣`
`outterm_4 = "hello world`	`"hello world`

Tabuľka 5.2: Použitie znakov `""` v definícii výstupného terminálu.

Sekcia Nonterminals

Sekcia *Nonterminals* slúži k deklarácii mien neterminálov danej gramatiky. Každé meno je deklarované na samostatnom riadku a musí začínať písmenom, za ktorým môžu nasledovať písmená alebo čísla. V tejto sekcii musí byť deklarovaný aj počiatočný neterminál zo sekcie *Language*. Veľkosť znakov nie je braná do úvahy, nakoľko generátor prevedie všetky písmená na ich veľkú variantu. Každá deklarácia neterminálu musí byť v rámci neterminálov danej gramatiky jedinečná.

Sekcia Actions

V Sekcii *Actions* sú definované jednotlivé sémantické akcie danej gramatiky. Definícia sémantickej akcie má nasledovnú formu:

```
[tag]; [access]; [return type]; [name]; [parameters]
code
<!end>
```

Prvý riadok definuje jednotlivé vlastnosti sémantickej akcie. Tieto vlastnosti sú od seba navzájom oddelené znakom `';'`. Uvedenie niektorých vlastností nie je povinné, avšak aj pri neuvedení niektorej vlastnosti je nutné uviesť oddeľujúci znak `';'`, aby bolo možné presne priradiť hodnoty k odpovedajúcim záznamom.

Tag slúži na identifikáciu akcie v rámci pravidiel danej gramatiky. Ak nie je akcia použitá v žiadnom pravidle, nie je nutné *tag* uvádzať vôbec. Ak je však

uvedený, je nutné, aby bol v rámci danej gramatiky jedinečný. Za prípustný tag je považovaný reťazec, ktorý začína písmenom a pokračuje písmenom, číslicou alebo znakom `'_'`.

Access špecifikuje viditeľnosť danej akcie z triedy vygenerovaného analyzátora. Prípustná množina hodnôt pre *access* je *{public, private}*. Pri špecifikácii ako *public* bude daná metóda viditeľná mimo triedy analyzátora, zatiaľ čo pri hodnote *private* nebude daná metóda viditeľná mimo triedy analyzátora.

Return type definuje návratový typ sémantickej akcie. Nakoľko návratovou hodnotou sémantickej akcie môže byť aj akýkoľvek užívateľsky definovaný typ, nie je možné efektívne testovať jej správnosť. Kontrola tohto záznamu sa teda obmedzuje len na prítomnosť aspoň jedného neprázdneho reťazca bez bielych znakov.

Name definuje názov metódy reprezentujúcej danú akciu vo vygenerovanom analyzátore. Za prípustné meno je považovaný reťazec, ktorý začína písmenom a pokračuje písmenom, číslicou alebo znakom `'_'`. Samotné meno akcie nie je analyzátorom nijako upravované.

Parameters definuje parametre sémantickej akcie. Ako hodnota tohto záznamu sa očakáva zoznam parametrov oddelený znakom `','`. Uvedenie hodnoty tohto záznamu nie je povinné, nakoľko sémantická akcia nemusí prijímať žiadne parametre. Podobne ako pri zázname *return type* môže hodnota tohto záznamu obsahovať akýkoľvek užívateľsky definovaný typ, a preto sa kontrola jednotlivých parametrov obmedzuje len na prítomnosť aspoň jedného neprázdneho reťazca bez bielych znakov.

Poznámka 5.3:
V skutočnosti sú pri načítavaní položiek v zozname parametrov sémanatickej akcie ignorované prázdne položky alebo položky obsahujúce len biele znaky. Spomínaná kontrola na prítomnosť aspoň jedného neprázdneho reťazca bez bielych znakov prebieha až pri spracovávaní akcie v rámci generátora.

Code predstavuje zdrojový kód akcie v jazyku *C++*. Tento kód je úplne prekopírovaný do tela akcie v implementácii vygenerovaného analyzátora a nie je nijako kontrolovaný. Za jeho správnosť zodpovedá užívateľ. Ak chce užívateľ vygenerovať len kostru akcie (teda definíciu s prázdnym telom), nemusí žiadny kód uvádzať.

Každá definícia sémantickej akcie musí byť ukončená špeciálnym reťazcom `<!end>`. Akékoľvek prázdne riadky (alebo komentáre) medzi definíciami jednotlivých akcií sú ignorované, avšak prázdne riadky a komentáre v definícii akcie sú kopírované do generovaného kódu.

Kvôli úplnosti ešte uvádzame niekoľko príkladov definície sémantických akcií aj s popisom:

Príklad 5.4:

```
a_print; private; void; print
out() << stack_[0];
<!end>
```

Tento príklad definuje nasledujúcu funkciu:

```
1 void print()
2 {
```

```
3   out() << stack_[0];
4 }
```

Listing 5.1: Príklad vygenerovanej metódy z definície sémantickej akcie.

Táto funkcia bude definovaná v privátnom rozhraní patričného analyzátora a volaná na hranách produkčných stromov odpovedajúcich hrane sémantickej akcie s tagom *a_print*. Je vhodné si všimnúť, že definícia akcie neobsahuje žiadne parametre. Položka pre parametre je na prvom riadku vždy uvedená ako posledná, a teda, ak akcia neprijíma žiadne parametre, nie je nutné uvádzať ani posledný oddeľovač `';'`.

Príklad 5.5:

```
; public; int; add; int a, int b
return a + b;
<!end>
```

Tento príklad definuje nasledujúcu funkciu:

```
1 int add(int a, int b)
2 {
3   return a + b;
4 }
```

Listing 5.2: Príklad vygenerovanej metódy z definície sémantickej akcie.

Táto funkcia bude definovaná vo verejnom rozhraní patričného analyzátora. Je dôležité si všimnúť, že metóda nemá definovaný *tag* (a teda jej definícia začína oddeľovačom `';'`).

Sekcia Action-includes

Sekcia *Action-includes* obsahuje zoznam hlavičkových súborov, ktoré sú potrebné k správnemu behu akcií. Každý riadok v tejto sekcii obsahuje meno jedného súboru, ktorý môže byť (pomocou direktívy *#include*) vložený buď do hlavičkového súboru alebo do súboru s implementáciou vygenerovaného parsera. Každé meno súboru v tejto sekcii je navyše prefixované podľa toho, kam má byť daný súbor vložený. Prefix `h:` značí vloženie do hlavičkového súboru, zatiaľ čo prefix `i:` značí vloženie do súboru s implementáciou. Samotné záznamy vyzerajú napríklad takto:

```
h:<vector>
i:"myactions.h"
```

Pri vkladaní definovaných hlavičkových súborov sú odstránené duplicitné záznamy. Poradie vložených hlavičkových súborov je zachované z konfiguračného súboru až na hlavičkové súbory potrebné k implementácii minimálnej funkčnosti vygenerovaného analyzátora, ktoré sú vždy uvedené ako prvé.

V hlavičkovom súbore vygenerovaného analyzátora predstavujú túto výnimku súbory `"parserutils.h"`, `<iostream>` a `<limits>`. V implementačnom súbore vygenerovaného analyzátora zase hlavičkový súbor vygenerovaného analyzátora, `<cctype>` a `<stdexcept>`.

Sekcia Action-members

V sekcii *Action-members* je možné deklarovať premenné, ktoré sú využívané akciami. Tieto deklarácie budú pridané k privátnym členským premenným vygenerovaného analyzátora. Deklarácia je vo forme *C++* kódu a za jej správnosť zodpovedá užívateľ.

Sekcia Action-init

Sekcia *Action-init* obsahuje zdrojový kód v jazyku *C++* slúžiaci k správnej inicializácii členských premenných deklarovaných v sekcii *Action-members*. Tento kód je vykonávaný v konštruktore triedy vygenerovaného analyzátora.

Sekcia Action-cleanup

Sekcia *Action-cleanup* obsahuje zdrojový kód v jazyku *C++* slúžiaci k správnemu uvoľneniu zdrojov alokovaných akciami. Tento kód je umiestnený v deštruktore triedy vygenerovaného parsera.

Sekcia Productions

V tejto sekcii sú definované všetky pravidlá, ktoré daná gramatika obsahuje. Pravidlo sa skladá z ľavej a pravej časti, ktoré sú od seba oddelené reťazcom znakov dve dvojbodky a znak rovná sa (`::=`). Zápis jednotlivých súčastí pravidla (neterminál, terminál, sémantická akcia alebo výstupný neterminál) sa riadi nasledujúcou konvenciou:

1. názov terminálu je zadávaný v okrúhlych zátvorkách (napríklad `(term)`),
2. názov neterminálu je zadávaný v hranatých zátvorkách (napríklad `[nterm]`),
3. názov výstupného terminálu je zadávaný v lomených zátvorkách (napríklad `<outterm>`),
4. názov sémantickej akcie je zadávaný v zložených zátvorkách (napríklad `{action}`).

V ľavej časti pravidla je vždy uvedený neterminál a v pravej časti je to kombinácia terminálov, neterminálov, výstupných symbolov a sémantických akcií. Pravá časť môže byť zároveň prázdna, čo znamená prepis na prázdne slovo. Pravá strana pravidla obsahujúceho ľavú rekurziu musí začínať príslušným neterminálom (z ľavej strany pravidla) - nesmie začínať výstupným neterminálom ani sémantickou akciou. Samotná definícia pravidla vyzerá napríklad takto:

```
[nterm]::=(term)[nterm]<out_term>{action}
```

Každé pravidlo je uvedené na samostatnom riadku. Všetky identifikátory jednotlivých položiek v pravidle musia byť už deklarované vo svojich príslušných sekciách.

Poznámka 5.6:
Poradie jednotlivých sekcií v rámci konfiguračného súboru nie je presne definované. Sekcie môžu byť usporiadané v ľubovoľnom poradí, avšak musí byť zachovaná podmienka o poradí deklarácií jednotlivých súčastí pravidla gramatiky pred ich použitím v danom pravidle.

Poznámka 5.7:
Jednotlivé sekcie môžu byť dokonca rozdelené alebo uvedené viackrát. Napríklad:

```
[Symbols]
    plus = +
    star = *

[OutTerminals]
    OutPlus = +
    OutMinus = -
    OutStar = *
    OutEqual = =

[Symbols]
    lpar = (
    rpar = )
    minus = -
```

je validná deklarácia symbolov.

5.3 Použitie aplikácie CppKind

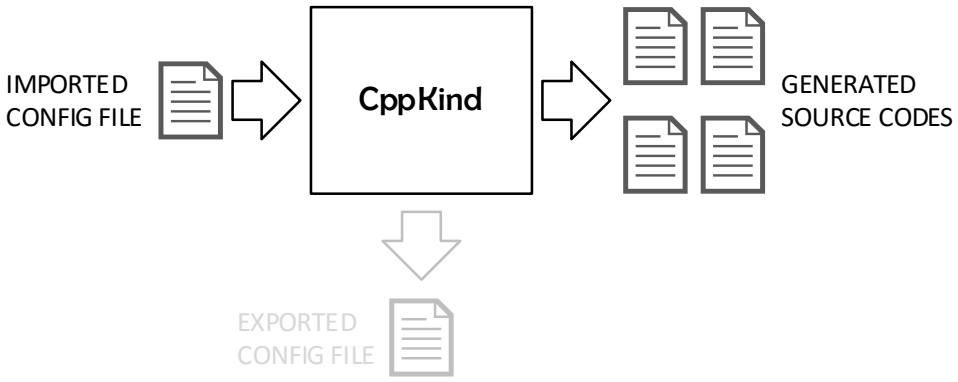
Aplikácia CppKind je spúšťaná na príkazovom riadku zavolaním príkazu `CppKind.exe`, ktorý má dva povinné argumenty:

`CppKind.exe <languageFile> <outputDirectory>`,
kde:

1. `<languageFile>` je konfiguračný súbor obsahujúci definíciu gramatiky a príznaky pre generátor,
2. `<outputDirectory>` je adresár, kde budú vygenerované súbory implementujúce vygenerovaný analyzátor.

Spustením tohto príkazu je načítaný konfiguračný súbor, z ktorého je získaný popis gramatiky a rôzne užívateľské nastavenia, ktoré ovplyvňujú zdrojový kód vygenerovaného analyzátora. Po načítaní tohto súboru sú vytvorené potrebné dátové štruktúry (produkčné stromy) a spočítané potrebné údaje (výhľady), z ktorých je následne možné vygenerovať analyzátor pre odpovedajúcu gramatiku.

V prípade, že všetko prebehne bez problémov, je program CppKind ukončený a v určenom adresári sú vytvorené zdrojové kódy vygenerovaného analyzátora. Tento proces je načrtnutý na obrázku 5.1.



Obr. 5.1: Diagram znázorňujúci vstupy a výstupy programu CppKind. Export konfiguračného súboru je popísaný v sekcii 5.4.

Ak počas behu programu dôjde k chybe, je na štandardný výstup vypísaná chybová správa obsahujúca popis chyby a program je ukončený.

Jednou z najčastejších chýb, s ktorou sa môže užívateľ pri používaní generátora CppKind stretnúť, je chyba v predanom konfiguračnom súbore. Chybová správa spojená s týmto druhom chyby môže mať napríklad nasledujúci tvar:

```
Invalid Language record: "casesensitive = "
```

Táto chybová správa odkazuje na nesprávny záznam v sekcii Language konfiguračného súboru, ktorý má obsah `"casesensitive = "` (problémom je chýbajúca hodnota v tomto zázname).

Počas načítania konfiguračného súboru môže generátor vypísať aj niekoľko varovaní. Varovanie slúži k upozorneniu užívateľa na prípadné komplikácie, ktoré ale neohrozujú generovanie analyzátora. Príkladom takéhoto varovania môže byť správa:

```
WARNING: string terminal in use, but no string delimiter was
specified. Using ´ " ´ as default.
```

Toto varovanie oznamuje, že užívateľ síce definoval terminál reprezentujúci znakový reťazec, ale nešpecifikoval oddeľovač znakových reťazcov, a preto bude použitá prednastavená hodnota `"`.

Po úspešnom spracovaní konfiguračného súboru je započaté samotné spracovanie gramatiky a výpočet potrebných dát. Počas tohto výpočtu môže byť zistené, že gramatika nespĺňa požiadavky kladené na prívetivé gramatiky. V tomto prípade je beh programu ukončený a je vypísaná chybová správa, ktorá môže mať napríklad tvar:

```
Grammar is not kind: NLRF(A) (intersect) DLRF(A) != emptyset.
```

Táto chybová správa oznamuje, že pre neterminál A nie je splnená prvá podmienka prívetivých gramatík.

Výpočet generátora môže byť ukončený kvôli niekoľkým ďalším chybám, ktoré nemá zmysel podrobne popisovať, nakoľko chybové hlášky pri výskyte týchto chýb sú dostatočne popisné.

5.4 Import a export nastavení

Ako už bolo spomínané, všetky nastavenia sú aplikácii CppKind predávané pomocou konfiguračného súboru, ktorý má presne definovanú štruktúru a položky, ktorých obsah vie aplikácia spracovať. Tento prístup však môže byť v niektorých prípadoch nevýhodný.

Jedným z týchto prípadov môže byť situácia, kedy užívateľ nemôže alebo nechce použiť preferovaný formát alebo spôsob spracovania. Uvažujme napríklad, že potrebné nastavenia má užívateľ rozdelené do viacerých súborov. Tento prístup je vhodný, ak má užívateľ väčšie množstvo gramatík, ktoré majú nejakú podmnožinu nastavení spoločnú, a preto je výhodnejšie mať ich uložené v samostatnom súbore.

Ďalším prípadom by mohli byť nastavania gramatiky uložené v relačnej databáze, kedy by mohlo byť pre užívateľa výhodnejšie ich načítavať priamo z tejto databázy namiesto použitia textového súboru.

Podobne môžu byť nastavenia uložené na nejakom vzdialenom mieste a užívateľ môže chcieť namiesto predávania lokálneho súboru používať URL vzdialeného zdroja.

Iným typom situácie, kedy je použitie preferovaného formátu nevhodné je, ak má užívateľ už vytvorené nastavenia gramatík, ktoré sú obsahovo kompatibilné s nastaveniami programu CppKind, ale ich formát nevyhovuje očakávanému formátu. Typickým príkladom by mohli byť konfiguračné súbory pre iné generátory analyzátorov.

Pretože podobných prípadov sa dá vymyslieť ľubovoľne veľa, nie je možné, aby bol každý podobný spôsob načítavania konfigurácie podporovaný. Namiesto toho je aplikácia CppKind navrhnutá tak, aby v prípade potreby mohol užívateľ aplikáciu rozšíriť o spôsob spracovania konfigurácie, ktorý momentálne potrebuje.

Dá sa povedať, že spracovanie nastavení z konfiguračného súboru v aplikácii CppKind je možné rozdeliť na tri fázy:

1. získanie nastavení,
2. zapísanie nastavení,
3. overenie nastavení.

Prvá fáza odpovedá prečítaniu záznamu v konfiguračnom súbore a následnej identifikácii položky nastavenia a jej hodnoty, ktorú záznam definuje. K tomu je nutné, aby podprogram vykonávajúci túto fázu poznal štruktúru konfiguračného súboru a jeho jednotlivých záznamov.

Druhá fáza má za úlohu nastaviť hodnotu získanú v prvej fáze do odpovedajúcich štruktúr používaných programom CppKind. K tomu je nutné, aby podprogram poznal tieto štruktúry a vedel, v akom formáte majú byť priradzované hodnoty.

Úlohou tretej fázy je skontrolovať, či boli spomínané štruktúry správne nastavené. To znamená, či im boli jednotlivé položky predané v správnom formáte s validnými hodnotami a či boli nastavené všetky položky potrebné k bezproblémovému behu aplikácie.

Pre ľahšie pochopenie si tento proces popíšeme na konkrétnom príklade. Predpokladajme, že podprogram vykonávajúci spracovanie konfiguračného súboru v prvej fáze načítal nasledujúci záznam:

```
casesensitive = no
```

K spracovaniu tohto záznamu musí vedieť, že konfiguračný súbor môže obsahovať nejaký záznam s kľúčom `casesensitive`, a že tento záznam má obsahovať hodnotu typu boolean. Hodnota `no` je však znakový reťazec, nie boolean, a preto musí podprogram ďalej vedieť ako získať hodnotu nastavenia v potrebnom formáte. V tomto prípade môže mať niekoľko prednastavených reťazcov s priradenou odpovedajúcou hodnotou typu boolean (napríklad znakový reťazec `no` môže odpovedať hodnote *false* typu boolean). Ak získaná hodnota neodpovedá žiadnemu z prednastavených reťazcov, je vhodné, aby podprogram ohlásil chybu.

Ďalšou fázou je nastavenie získanej hodnoty do odpovedajúcej štruktúry v programe CppKind. Touto štruktúrou je trieda *Grammar*, ktorá predstavuje v rámci programu gramatiku a sú v nej počas behu uchované všetky nastavenia. Táto trieda má presne definované rozhranie, pomocou ktorého je možné narábať s jej inštanciami. Konkrétne pre nastavenie položky case-sensitive poskytuje trieda *Grammar* metódu *setCaseSensitive(bool enabled)*. Podprogram teda môže použiť túto funkciu k nastaveniu hodnoty získanej z konfiguračného súboru.

V poslednej tretej fáze prebieha kontrola predaných nastavení. V tomto konkrétnom prípade sa žiadna kontrola nevykonáva, keďže formát predanej hodnoty je vždy správny (metóda *setCaseSensitive* prijíma len hodnoty typu boolean). Pri nastavovaní iných položiek (napríklad tvaru symbolov) je však vhodné kontrolovať predanú hodnotu priamo v metóde, ktorá sa o nastavovanie stará. Po ukončení načítavania nastavení je rovnako vhodné skontrolovať, či boli nastavené všetky potrebné položky.

Z tohto príkladu je vidieť, že v programe CppKind je možné implementovať druhú a tretiu fázu staticky v triede *Grammar*. Pretože položky nastavení, ktoré program očakáva z konfiguračného súboru sú vždy rovnaké, môže trieda *Grammar* implementovať rozhranie, ktoré dovolí všetky tieto položky nastaviť a skontrolovať ich validitu. Toto jednotné rozhranie môže byť využité podprogramom, ktorý spracováva konfiguračný súbor.

S využitím tohto rozhrania sa celý problém načítavania nastavení redukuje na vytvorenie mechanizmu, ktorý by využitím rozhrania triedy *Grammar* nastavil práve používanú inštanciu tejto triedy. K implementácii tohto mechanizmu v rámci aplikácie CppKind slúži rozhranie triedy *Importer*.

Trieda *Importer* je abstraktná trieda, ktorá deklaruje nasledujúcu čiste virtuálnu funkciu:

```
virtual void importGrammar(Grammar *) = 0
```

Táto funkcia dostáva ako parameter ukazovateľ na objekt triedy *Grammar*. Predpokladá sa, že trieda odvodená od triedy *Importer* implementuje v tejto funkcii spôsob načítania jednotlivých nastavení a ich prevod do formátu použiteľného s rozhraním triedy *Grammar*, ktoré následne použije k nastaveniu predaného objektu. Týmto vznikne objekt schopný nastaviť inštanciu triedy *Grammar* z ľubovoľného zdroja ľubovoľným spôsobom.

Metódu *importGrammar* chceme používať len v spojení s triedou *Grammar*, a preto je táto metóda v triede *Importer* označená ako chránená (protected) a trieda *Grammar* je označená za priateľskú triedu (friend class) triedy *Importer*.

Trieda *Grammar* poskytuje nasledujúcu metódu:

```
void importSettings(Importer * , bool)
```

Táto metóda slúži na nastavenie inštancie triedy *Grammar* pomocou predaného ukazovateľa na objekt typu *Importer*. Druhý parameter je nepovinný (má prednastavenú hodnotu *true*) a určuje, či majú byť pred použitím importéru nastavené východzie hodnoty.

Načítanie konfiguračných hodnôt prebieha v aplikácii CppKind nasledovným postupom:

1. vytvorí sa inštancia `g` triedy *Grammar*,
2. vytvorí sa inštancia `importer` triedy implementujúcej rozhranie triedy *Importer*,
3. na inštancii `g` sa zavolá metóda *importSettings*, ktorej sa predá ukazovateľ na `importer` a príznak signalizujúci, či majú byť obnovené východzie nastavenia,
   (a) ak je nastavený príznak, tak sa v `g` nastavia východzie nastavenia (ako keby bolo `g` práve vytvorené),
   (b) cez predaný ukazovateľ na `importer` sa zavolá metóda *importGrammar*, ktorej je predaný ukazovateľ na `g`,
   (c) v metóde *importGrammar* sa pomocou predaného ukazovateľa nastavia všetky potrebné nastavenia,
4. objekt `g` je nastavený a pred použitím sa vykoná overenie, že boli nastavené všetky nutné položky.

Príklad 5.8 poskytuje jednoduchú ukážku implementácie metódy *importGrammar* triedy *Importer*, ktorá slúži na nastavenie jednoduchej statickej gramatiky.

Príklad 5.8 (Jednoduchá implementácia metódy *importGrammar*):

```
1  void ExampleLoader::importGrammar(Grammar * g) {
2    try {
3      // lookahead
4      g->setLookahead(3);
5      // case sensitivity
6      g->setCaseSensitive(false);
7      // view trees mode
8      g->setViewMode(ViewTreeMode::depthMinimal);
9      // namespace
10     g->addNamespace("Example");
11     // parser class name
12     g->setParserClassName("ExampleAnalyzer");
13     // symbols
14     g->addSymbol("a", "a");
15     g->addSymbol("b", "b");
16     g->addSymbol("c", "c");
17     g->addSymbol("d", "d");
18     // nonterminals
19     g->addNonterminal("S", true);
20     g->addNonterminal("A");
21     g->addNonterminal("B");
22     g->addNonterminal("C");
```

```
23
24      typedef ProductionItemType PIT;
25      ProductionInfo p;
26      //S -> A
27      p.push_back(ProdItemInfo("S", PIT::nonterminal));
28      p.push_back(ProdItemInfo("A", PIT::nonterminal));
29      g->addProduction(p);
30      p.clear();
31
32      // A -> Ba
33      p.push_back(ProdItemInfo("A", PIT::nonterminal));
34      p.push_back(ProdItemInfo("B", PIT::nonterminal));
35      p.push_back(ProdItemInfo("a", PIT::terminal));
36      g->addProduction(p);
37      p.clear();
38
39      // A -> Cd
40      p.push_back(ProdItemInfo("A", PIT::nonterminal));
41      p.push_back(ProdItemInfo("C", PIT::nonterminal));
42      p.push_back(ProdItemInfo("d", PIT::terminal));
43      g->addProduction(p);
44      p.clear();
45
46      // B -> Cb
47      p.push_back(ProdItemInfo("B", PIT::nonterminal));
48      p.push_back(ProdItemInfo("C", PIT::nonterminal));
49      p.push_back(ProdItemInfo("b", PIT::terminal));
50      g->addProduction(p);
51      p.clear();
52
53      // C -> c
54      p.push_back(ProdItemInfo("C", PIT::nonterminal));
55      p.push_back(ProdItemInfo("c", PIT::terminal));
56      g->addProduction(p);
57      p.clear();
58
59    }
60    catch (CppKindException& ex) {
61      std::cout << ex.what() << std::endl;
62    }
63    catch (std::exception& ex) {
64      std::cout << ex.what() << std::endl;
65    }
66  }
```

Listing 5.3: Ukážka jednoduchej implementácie metódy *importGrammar*.

Poznámka 5.9:
Popis rozhrania triedy *Grammar* v tomto dokumente neuvádzame. Tento popis je možné nájsť v programovej dokumentácii projektu CppKind, ktorá sa nachádza na priloženom CD.

Poznámka 5.10:
V prípade potreby komplexnej ukážky načítavania vstupu pomocou importéru môže užívateľ konzultovať implementáciu triedy *CppKindImporter*, ktorá je v projekte využívaná k načítavaniu konfiguračného súboru v štandardnom formáte. Táto implementácia umožňuje import všetkých dostupných nastavení, a preto môže

slúžiť ako cenný zdroj informácií pri vytváraní vlastného importéru. Implementácia tejto triedy sa nachádza v súboroch *cppkindloader.h* a *cppkindloader.cpp*

Poznámka 5.11:
Ak chce užívateľ použiť vlastnú implementáciu importéru, musí zároveň upraviť funkciu *main* definovanú v súbore *app.cpp* tak, aby využívala ním definovaný importér namiesto štandardného importéru *CppKindImporter*. Ak kvôli tomu potrebuje zmeniť argumenty predávané z príkazového riadku, musí zabezpečiť, aby boli ostatným súčastiam programu predané správne hodnoty (napr. adresár kam budú vygenerované zdrojové kódy).

Okrem importu nastavení má užívateľ možnosť tieto nastavenia aj exportovať. Za exportovanie nastavení sa považuje situácia, kedy užívateľ z už nastaveného objektu triedy *Grammar* vyextrahuje množinu nastavení, ktoré tento objekt obsahuje a uloží ich do formátu, pomocou ktorého je možné nastaviť iný objekt triedy *Grammar* tak, že oba objekty budú mať zhodné nastavenia.

Tento formát nemusí byť nutne zhodný s formátom, ktorý bol použitý k nastaveniu prvého objektu. V praxi to znamená, že je možné urobiť prevod nejakého formátu konfigurácie na iný formát.

Mechanizmus exportu nastavení je v aplikácii CppKind riešený podobne ako mechanizmus importu nastavení a k jeho implementácii slúži trieda *Exporter*. Táto trieda deklaruje čiste virtuálnu funkciu:

```
virtual void exportGrammar(Grammar *) = 0
```

Táto funkcia dostáva ako parameter ukazovateľ na objekt typu *Grammar*. Predpokladá sa, že trieda odvodená od triedy *Exporter* implementuje prevod nastavení získaných z predaného objektu do požadovaného formátu. Nastavenia je z objektu možné získať použitím rozhrania triedy *Grammar*.

Podobne ako pri importe ani pri exporte nechceme, aby bolo možné volať priamo funkciu *exportSettings*, a preto je táto funkcia definovaná ako chránená a trieda *Grammar* uvedená ako priateľská trieda triedy *Exporter*.

K použitiu tohto mechanizmu definuje trieda *Grammar* nasledujúcu funkciu:

```
void exportSettings(Exporter *)
```

Táto metóda použije predaný ukazovateľ na objekt triedy *Exporter* k vyexportovaniu svojich aktuálnych nastavení.

Export nastavení z objektu triedy *Grammar* teda prebieha nasledovným spôsobom:

1. nech `g` je inštancia triedy *Grammar* s už naimportovanými nastaveniami,
2. vytvorí sa inštancia *exporter* triedy implementujúcej rozhranie triedy *Exporter*,
3. na inštancii `g` sa zavolá metóda *exportSettings*, ktorej sa ako parameter predá ukazovateľ na `exporter`,
   (a) cez predaný ukazovateľ na `exporter` sa zavolá metóda *exportGrammar*, ktorej je predaný ukazovateľ na `g`,
   (b) v metóde *exportGrammar* sa z predaného ukazovateľa pomocou rozhrania triedy *Grammar* vyextrahujú všetky nastavenia a prevedú sa do požadovaného formátu.

Poznámka 5.12:
V prípade potreby komplexnej ukážky exportu nastavení z objektu triedy *Grammar* môže užívateľ konzultovať implementáciu triedy *CppKindExporter*, ktorá definuje export nastavení do štandardného formátu používaného aplikáciou CppKind. Implementácia tejto triedy sa nachádza v súboroch *cppkindloader.h* a *cppkindloader.cpp*.

Poznámka 5.13:
Keďže v základnej forme používa aplikácia CppKind len štandardný konfiguračný súbor, nie je dôvod dodaný exportér v aplikácii používať. V prípade, že ale užívateľ definuje vlastný importér, je možné použiť dodaný exportér k vytvoreniu konfiguračného súboru v štandardnom formáte. K tomu je nutné upraviť metódu *main* definovanú v súbore *app.cpp*.

5.5 Prehľad súborov

Pre úplnosť uvádzame zoznam zdrojových súborov tvoriacich aplikáciu CppKind spolu s krátkym popisom ich obsahu.

- action.h, action.cpp

  Spracovanie sémantických akcií.

- app.cpp

  Súbor s definícou funkcie `main`.

- constants.h

  Definícia konštánt využívaných v aplikácii.

- cppkindloader.h, cppkindloader.cpp

  Implementácia importu a exportu konfiguračného súboru.

- debug.h

  Direktívy preprocesoru spojené s debugovaním.

- filegenerator.h, filegenerator.cpp

  Abstrakty generátorov zdrojových súborov.

- global.h

  Globálne definície.

- grammar.h, grammar.cpp

  Implementácia štruktúry udržiavajúcej informácie o gramatike.

- lookahead.h

  Pomocné funkcie na prácu s výhľadmi.

- nonterminal.h, nonterminal.cpp

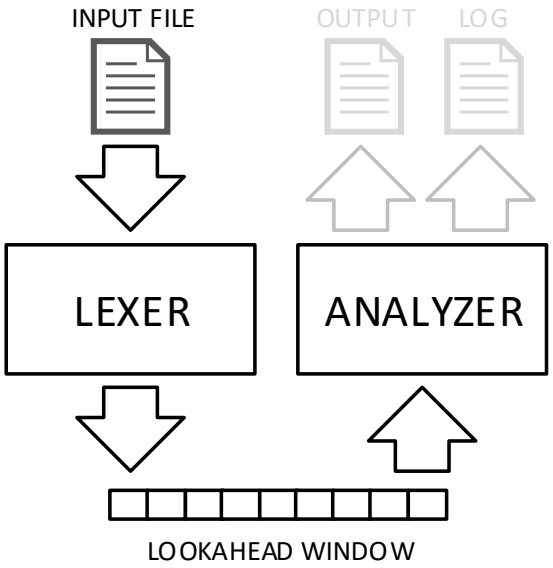
  Implementácia štruktúry neterminálov.

- parserfiles.h, parserfiles.cpp
  Implementácia generátora zdrojových súborov analyzátora.
- printer.h, printer.cpp
  Implementácia pomocnej štruktúry na prácu s výstupom.
- prodtree.h, prodtree.cpp
  Implementácia produkčných stromov.
- production.h, production.cpp
  Implementácia pravidiel gramatiky.
- stack.h
  Implementácia generickej štruktúry zásobníka.
- symtree.h, symtree.cpp
  Implementácia štruktúry symbolických stromov.
- terminal.h, terminal.cpp
  Implementácia štruktúry terminálov.
- util.h, util.cpp
  Pomocné funkcie.
- utilfiles.h, utilfiles.cpp
  Implementácia generátora statickej časti analyzátora.
- viewcalc.h, viewcalc.cpp
  Implementácia výpočtu výhľadových množín a výhľadov.
- viewtree.h, viewtree.cpp
  Implementácia výhľadových stromov.

Kapitola 6

Návrh a štruktúra analyzátora

Cieľom nasledujúcej kapitoly je poskytnúť hlbší prehľad o štruktúre vygenerovaného analyzátora (ďalej len analyzátora), popísať jeho jednotlivé súčasti a ich vzájomnú interakciu. Ďalej by mala priblížiť jednotlivé triedy, ktoré vytvárajú analyzátor a objasniť niektoré rozhodnutia učinené pri návrhu týchto tried, čím by mala užívateľovi uľahčiť prácu pri prípadnej nutnosti modifikácie analyzátora.



Obr. 6.1: Proces analýzy súboru so vstupným textom.

Obrázok 6.1 ilustruje základné časti vygenerovaného analyzátora (translátora) použité pri analýze súboru so vstupným textom (*Input file*). Ten je predaný vygenerovanému analyzátoru (translátoru) a spracovávaný lexerom (*Lexer*), ktorý v ňom vyhľadáva inštancie terminálov gramatiky. Tie sú v podobe tokenov ukladané do výhľadového okna (*Lookahead window*). Z tokenu je možné zistiť odpovedajúci terminál a pri niektorých tokenoch (tam kde je to vhodné) aj sémantickú hodnotu v podobe atribútu získaného zo vstupného textu. Syntaktický analyzér (*Analyzer*) následne prevádza syntaktickú (a v prípade translátora aj sémantickú) analýzu na základe tokenov vo výhľadovom okne. Pri svojej činnosti môže analyzér na základe nastavenia vytvárať log (*Log*) a v prípade translátora aj výstup (*Output*) generovaný výstupnými terminálmi a aplikáciou sémantických akcií.

Poznámka 6.1:
Keďže pomenovanie niektorých častí analyzátora je definované užívateľom, budeme v nasledujúcom texte predpokladať, že trieda predstavujúca samotný analyzátor je pomenovaná Analyzer. Ďalej budeme predpokladať, že táto trieda nie je umiestnená v žiadnom mennom priestore (ak ho explicitne neuvedieme), aj napriek tomu, že užívateľ môže tento menný priestor definovať. Všetky ostatné názvy dátových štruktúr v nasledujúcom texte odpovedajú ich pomenovaniu vo vygenerovanom zdrojovom kóde.

Poznámka 6.2:
Názvy jednotlivých menných priestorov, tried, vymenovaných typov, metód a premenných prevzatých priamo z vygenerovaného kódu budeme označovať kurzívou (napríklad *Analyzer*).

6.1 Členenie analyzátora

Jednotlivé triedy tvoriace analyzátor je možné rozdeliť do dvoch skupín. Prvá skupina predstavuje statickú časť analyzátora, zatiaľ čo druhá skupina prestavuje dynamickú časť.

Statickú časť tvoria triedy navrhnuté tak, aby ich implementácia vyhovovala každému analyzátoru. Tieto triedy sú definované v súboroch `_parserutils.h` a `_parserutils.cpp` . Do tejto skupiny patria nasledujúce triedy:

1. *AttributeType*
2. *Attribute*
3. *Token<T>*
4. *RingBuffer<T>*
5. *Input*
6. *LexicalAnalyzer<T>*
7. *lexical_error*
8. *syntax_error*
9. *internal_error*

Dynamickú časť tvoria triedy, ktorých implementácia je špecifická pre každý analyzátor. Ich implementácia je definovaná v súboroch, ktorých názov je závislý od pomenovania triedy reprezentujúcej analyzátor - spravidla začína tento názov podčiarkovníkom, za ktorým nasleduje názov triedy analyzátora tvorený malými písmenami a odpovedajúcou príponou (.h alebo .cpp). Teda za predpokladu, že trieda predstavujúca analyzátor je pomenovaná *Analyzer*, bude implementácia definovaná v súboroch `_analyzer.h` a `_analyzer.cpp` . Do dynamickej skupiny patria nasledujúce triedy:

1. *Terminal*

2. *Lexer*
3. *Analyzer*
4. *Tracer*

Rozdelenie implementácie analyzátora na statickú a dynamickú časť má niekoľko výhod, napríklad:

1. možnosť opakovaného použitia statickej časti pri použití viacerých analyzátorov,
2. schopnosť vytvárať súbory analyzátorov pre rôzne gramatiky, avšak s použitím jednotného návrhu implementácie,
3. jednoduchšia údržba,
4. možnosť špecificky upravovať konkrétny analyzátor.

6.2 Prehľad jednotlivých tried

V nasledujúcej sekcii sa budeme venovať jednotlivým triedam, ktoré tvoria analyzátor. Začneme popisom vymenovaného typu *Terminal*, ktorý patrí k dynamickej časti analyzátora, pretože tento vymenovaný typ ovplyvňuje hneď niekoľko tried zo statickej časti. Následne predstavíme triedy tvoriace statickú časť a sekciu uzavrieme popisom tried zabezpečujúcich lexikálnu a syntaktickú analýzu.

6.2.1 Terminal

Ako už bolo uvedené, *Terminal* je vymenovaný typ, konkrétne silne typovaný vymenovaný typ (`enum class`). Tento typ je súčasťou dynamickej časti analyzátora, čo znamená, že jeho obsah je dynamicky generovaný a pre každý vygenerovaný analyzátor je jedinečný. *Terminal* slúži k definícii identifikátorov jednotlivých terminálov z gramatiky popisujúcej daný analyzátor. Hodnoty tohto typu sú vytvorené z názvov terminálov v konfiguračnom súbore, pričom znaky v každom názve sú prevedené na ich veľkú variantu a celý názov je prefixovaný dvojicou znakov `"T_"`. Teda napríklad pre terminál s názvom *plus* bude vygenerovaná hodnota `Terminal::T_PLUS`, ktorá bude predstavovať identifikátor daného terminálu v rámci celého analyzátora.

Použitie prefixu pri vytváraní hodnoty je dôležité, pretože nám umožňuje pridávať implicitné hodnoty bez toho, aby sme museli obmedzovať užívateľa pri pomenovávaní ním definovaných terminálov - pre implicitné hodnoty proste použijeme iný prefix. Toto je hneď využité pri definícii dvoch špeciálnych hodnôt, ktoré vymenovaný typ *Terminal* vždy implicitne obsahuje. Týmito hodnotami sú `Terminal::S_EOI` a `Terminal::S_NOTERM`.

Prvá z uvedených hodnôt slúži ako identifikátor terminálu pre koniec vstupu. Druhá hodnota v skutočnosti nie je identifikátorom žiadneho terminálu, ale slúži ako návratová hodnota v prípade, že nie je dostupný žiadny iný identifikátor terminálu.

K implementácii bol zvolený silne typovaný vymenovaný typ (`enum class`) namiesto tradičného vymenovaného typu (`enum`) z nasledujúcich dôvodov:

1. tradičný vymenovaný typ sa implicitne konvertuje na typ `int` - pri prevádzaní analýzy sa silne spolieha na porovnávanie hodnôt tohto typu, čo by mohlo pri úprave kódu užívateľom viesť k neočakávaným a ťažko detekovateľným problémom a chybám,

2. tradičný vymenovaný typ zavádza jeho hodnoty aj do okolitého menného priestoru, čo by mohlo viesť k potenciálnym kolíziám mien pri budúcom rozširovaní zdrojového kódu.

Ďalšou zvažovanou možnosťou implementácie bolo použitie triedy (*class*), kde by jednotlivé hodnoty boli reprezentované a konštanty definované v danej triede, napr:

```
static const int T_PLUS = 1;
```

Výhodou tejto reprezentácie by bola možnosť vytvoriť jednu základnú triedu terminálov, ktorá by obsahovala len implicitné hodnoty a všetky ostatné hodnoty by boli pridávané pomocou dedičnosti v derivovaných triedach špecializovaných pre jednotlivé analyzátory. To by zároveň umožnilo obmedziť typ šablónových parametrov pri niektorých triedach zo statickej časti, čo nie je možné pri vymenovaných typoch. Rovnako by to umožnilo pridávanie metód priamo spojených s identifikátormi terminálov (príkladom môže byť metóda toString(), ktorá prevedie identifikátor na jeho reťazcovú podobu), to je však možné zabezpečiť aj inými spôsobmi.

Nakoniec však prevládol prístup využívajúci vymenovaný typ vďaka jeho jednoduchosti a prehľadnosti.

Vymenovaný typ *Terminal* je vnorený v triede *Analyzer*, pretože to najlepšie odpovedá charakteru tohto typu - je jedinečne špecializovaný pre daný analyzátor a jeho použitie v inom analyzátore nie je vhodné. Zároveň to umožnuje použitie tohto mena v triedach iných analyzátorov bez nutnosti umiestňovať triedy jednotlivých analyzátorov do rôznych menných priestorov.

6.2.2 AttributeType

AttributeType je silne typovaný vymenovaný typ, ktorého hodnoty (ako názov napovedá) popisujú typ atribútu. Tieto hodnoty sú nasledovné:

1. *atInt* - reprezentuje celé číslo,

2. *atReal* - reprezentuje desatinné číslo,

3. *atChar* - reprezentuje znak,

4. *atString* - reprezentuje znakový reťazec,

5. *atId* - reprezentuje identifikátor,

6. *atNone* - reprezentuje typ, ktorý nie je známy alebo jeho znalosť nie je podstatná.

Hodnoty, ktoré tento typ poskytuje sú vybrané tak, aby odpovedali čo najvšeobecnejšiemu rozdeleniu hodnôt, s ktorými sa môže lexikálny analyzátor pri svojej práci stretnúť. To však nemusí byť vždy dostačujúce - v tom prípade by mal užívateľ doplniť hodnotu, ktorá bude chýbajúci typ popisovať.

Poznámka 6.3:
Je dôležité si uvedomiť, že aj keď dané hodnoty nie vždy odpovedajú požadovanému typu, často nie je nutné pridávať novú hodnotu reprezentujúcu daný typ. Napríklad na reprezentáciu hodnotového typu boolean je možné (znovu) využiť hodnotu *atInt*. Rovnako je možné reprezentovať zložitejšie typy pomocou kombinácie niekoľkých už existujúcich hodnôt - napríklad namiesto zavedenia hodnoty *atDate* je možné získať v sémantickej analýze hodnotu zložením troch atribútov s typovými hodnotami *atInt* pre reprezentáciu dňa, mesiaca a roku.

K reprezentácii typu *AttributeType* bol silne typovaný vymenovaný typ vybraný z rovnakých dôvodov pri type *Terminal*. Na rozdiel od typu *Terminal* je však typ *AttributeType* súčasťou statickej časti, a preto môže byť zdieľaný medzi viacerými analyzátormi. Pri pridaní hodnôt do tohto typu by nemali vzniknúť žiadne problémy, avšak odobratím nejakej hodnoty k problémom dôjsť môže, preto nie je vhodné hodnoty odoberať (radšej len nechať nevyužité).

6.2.3 Attribute

Trieda *Attribute* predstavuje atribút nejakého tokenu získaného pri lexikálnej analýze. Jej štruktúra je veľmi jednoduchá, v podstate slúži na uchovanie lexému, ktorý bol identifikovaný pri lexikálnej analýze tokenu a hodnoty (*AttributeType*), ktorá popisuje typ daného atribútu. Trieda preto poskytuje nasledujúce dve metódy:

- *type()* - vracia hodnotu popisujúcu typ atribútu (*AttributeType*),
- *lexeme()* - vracia samotný lexém identifikovaný pri lexikálnej analýze vo forme znakového reťazca (`std::string`).

Pri návrhu tejto triedy bolo dôležité uvedomiť si, ako sa bude v rámci analyzéru pracovať s atribútmi. Do úvahy pripadala možnosť implementovať túto časť pomocou typu `union`, ktorý zjednodušene povedané umožňuje reprezentovať pomocou jednej štruktúry viacero typov súčasne (ale len jeden aktívne) za použitia veľkosti pamäti odpovedajúcej najväčšiemu reprezentovanému typu. Druhou možnosťou bolo použiť už popísaný spôsob, ktorý predstavuje trieda *Attribute*.

Dôvodom k použitiu triedy *Attribute* oproti typu `union` bol spôsob spracovania atribútov analyzérom. Ten v podstate atribúty len uchováva a nesnaží sa ich nijako použiť. Použitie hodnôt jednotlivých atribútov je definované výhradne užívateľom v sémantických akciách. Z tohto dôvodu má trieda *Attribute* niekoľko výhod a to hlavne pri reprezentácii typu, ktorý očakáva užívateľ.

Pri použití typu `union` je nutné špecifikovať všetky dátové typy, ktoré by užívateľ mohol potrebovať a zároveň by rovnako bola nutná nejaká obdoba typu *AttributeType*, ktorá by určovala, čo je momentálne v premennej typu `union` uložené. Rovnako by muselo implicitne dochádzať k pretypovaniu na odpovedajúci typ, zatiaľ čo pri použití triedy *Attribute* sa uloží len reťazcová forma lexému a o pretypovanie sa stará užívateľ až v momente, keď je to potrebné.

Pri niektorých dátových typoch naviac môže byť použitie typu `union` neúmerne komplikované. Predstavme si napríklad, že užívateľ potrebuje reprezentovať veľmi dlhé čísla, ktorým neodpovedá žiadny štandardne definovaný dátový typ, a preto používa vlastný. Ak by boli atribúty reprezentované pomocou `union`, musel by byť tento typ pridaný do definície, čo by predstavovalo nie len ďalší nutný zásah do statickej časti analyzátora, ale aj možné zvýšenie pamäťových nárokov, ak by bol daný typ väčší ako všetky ostatné typy použité v type `union`.

Ďalšou výhodou použitia triedy *Attribute* je odovzdanie zodpovednosti za správnu reprezentáciu výsledného typu užívateľovi. To znamená, že sa analyzátor nesnaží rozhodovať aký typ použiť k uloženiu pretypovanej hodnoty. Uvažujeme napríklad atribút, v ktorom je uložené celé číslo (typ *AttributeType::atInt*). Na uloženie tohto celého čísla je možné použiť hneď niekoľko štandardných typov od `char` až po `long long` a namiesto toho, aby sme hádali, ktorý máme použiť - alebo použili hneď ten najväčší - nechávame rozhodnutie na užívateľovi, ktorý môže vybrať správny typ na základe kontextu (v definícii gramatiky).

Poznámka 6.4:
Nie všetky atribúty ukladajú odpovedajúci lexém. Ide konkrétne o atribúty tokenov, ktorých lexém je jednoducho odvoditeľný alebo jeho presná znalosť nie je potrebná. Sem patria atribúty tokenov kľúčových slov a symbolov, kedže tie majú presne definovaný tvar. Atribúty týchto tokenov majú štandardne typ *AttributeType::atNone* a hodnota lexému odpovedá prázdnemu reťazcu (`""`).

Poznámka 6.5:
Hodnoty lexému v atribútoch, ktoré majú typ *AttributeType::atChar* a *AttributeType::atString* (atribúty reprezentujúce znaky a znakové reťazce) sú ukladané bez ohraničujúcich oddeľovačov, a teda, ak sa počas lexikálnej analýzy v texte narazí na znak vo forme `'c'` (kde `'` je oddeľovač), tak odpovedajúca hodnota lexému v atribúte je `"c"` namiesto hodnoty `"'c'"`.

Fakt, že trieda *Attribute* nie je závislá na žiadnom subjekte z dynamickej časti analyzátora dovoľuje voľnú výmenu atribútov medzi viacerými analyzátormi, bez nutnosti nejakej konverzie.

V súvislosti s návrhom triedy *Attribute* treba ešte spomenúť, že konštruktor tejto triedy je privátny. Dôvodom k tomuto kroku, podobne ako pri ďalších triedach je fakt, že celkovo je parser navrhnutý ako celok, ktorý môže byť chápaný ako časť nejakej knižnice, a preto je vhodné obmedziť možnosť vytvárať inštancie tejto triedy len na subjekty, ktoré túto možnosť nevyhnutne potrebujú. Týmito triedami sú *Token* a *LexicalAnalyzer* a v návrhu sú označené ako priateľské triedy (`friend class`). Užívateľ sám by nemal mať potrebu vytvárať nové inštancie tejto triedy a mal by si vystačiť s využívaním tých, ktoré mu predáva lexikálny analyzátor.

6.2.4 Token

Trieda *Token* reprezentuje token ako produkt lexikálnej analýzy. Jej úlohou je asociovať patričný identifikátor terminálu z typu *Terminal* s odpovedajúcim atribútom (inštanciou triedy *Attribute*).

Táto trieda je implementovaná ako šablónová trieda s jedným šablónovým parametrom T. Pri špecializovaní tejto triedy sa predpokladá, že za parameter T

bude dosadený typ *Terminal*, čím vznikne špecializovaná trieda schopná reprezentovať tokeny odpovedajúce danej množine terminálov. Trieda *Token* poskytuje dve metódy:

1. terminal() - vracia identifikátor terminálu - hodnotu typu T (predpokladá sa, že T bude typ *Terminal* nejakej konkrétnej gramatiky),

2. attribute() - vracia atribút terminálu ako inštanciu triedy *Attribute*.

Trieda bola navrhnutá s použitím šablón, aby bola uľahčená jej prípadná údržba. Parametrizovanie triedy nám umožňuje mať len jednu definíciu a tú potom prispôsobovať pre konkrétnu množinu terminálov. Ak by sme šablónovú variantu nepoužili, musela by byť táto trieda implementovaná ako vnorená trieda pre každý analyzátor zvlášť a v prípade potreby by musela byť upravená každá z týchto vnorených tried.

Z podobných dôvodov ako pri triede *Attribute* je konštruktor triedy *Token* privátny. Navyše tým získavame určitú kontrolu nad tým, s akým šablónovým parametrom bude trieda *Token* použitá. Keďže prístup ku konštruktoru tejto triedy majú len spriatelené triedy *RoundBuffer* a LexicalAnalyzer, je tento parameter možné jednoducho obmedziť.

Poznámka 6.6:
Čitateľovi by sa mohla naskytnúť otázka, prečo nie je *Token* a *Attribute* implementovaný ako jedna trieda, ktorá by priamo obsahovala identifikátor terminálu, typ atribútu a lexém. Dôvod je jednoduchý - typ *Terminal* je viditeľný len vo vnútri analyzátora, ktorý ho využíva, preto je pre umožnenie predávania atribútov medzi analyzátormi dôležité, aby *Attribute* nebol závislý na žiadnej dynamickej súčasti analyzátora, čo viedlo k vzniku týchto dvoch tried.

Vzťah doteraz predstavených tried ilustruje obrázok 6.2.

6.2.5 Input

Pri lexikálnej analýze je vhodné mať nástroj, ktorý umožní jednoducho pracovať so vstupným textom. Vstupný text je počas lexikálnej analýzy spracovávaný sekvenčne zľava doprava. Občas je však vhodné mať možnosť sa v danom texte vrátiť o niekoľko znakov späť alebo naopak nahliadnuť za práve spracovávaný znak. V našom lexikálnom analyzátore túto funkcionalitu poskytuje trieda *Input*.

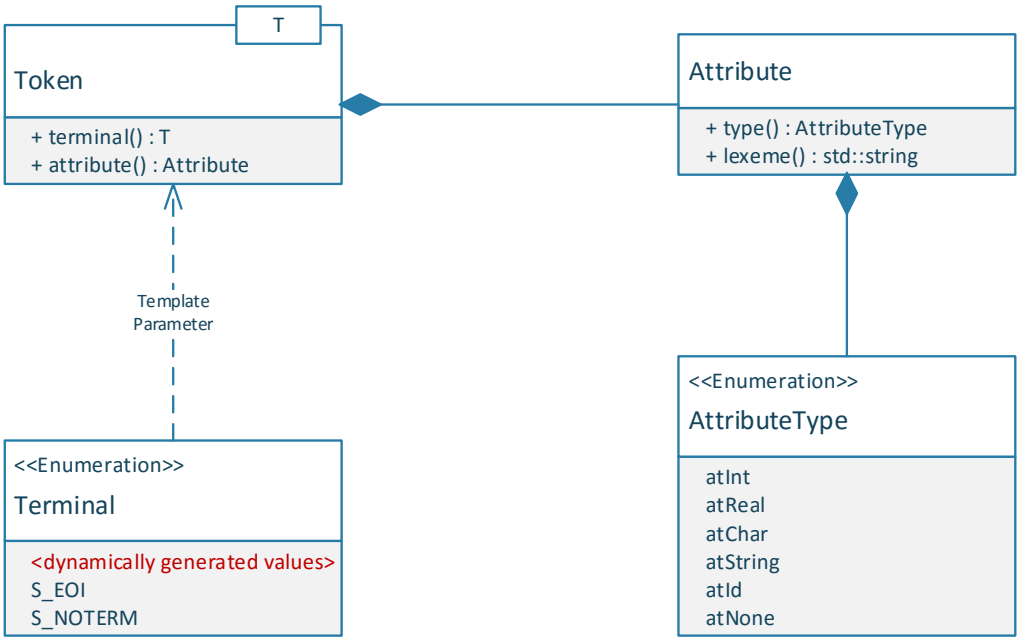
Trieda *Input* implementuje bufer, do ktorého ukladá načítavaný text zo vstupného súboru a ten potom poskytuje pomocou svojho rozhrania lexikálnemu analyzátoru. Pri implementácii je braný ohľad na efektívnosť potrebných operácii a typ použitý k implementácii buferu musí spĺňať nasledujúce podmienky:

1. zachovanie poradia pridávaných znakov,

2. rýchle pridávanie prvkov na koniec buferu,

3. rýchle odoberanie prvkov zo začiatku buferu,

4. rýchly prístup k n-tému prvku v buferi.



Obr. 6.2: UML diagram popisujúci vzťah typov Terminal, AttributeType a tried Attribute a Token.

Všetky tieto podmienky spĺňa typ `std::deque` zo štandardnej knižnice jazyka *C++*. K reprezentácii znakov zo vstupu bol zvolený typ `int` namiesto typu `char`, pretože je nutné reprezentovať aj symbol pre koniec vstupu (`EOF`), ktorý má v jazyku *C++* hodnotu -1, čo je aj dôvod, prečo je typ `int` návratovým typom niektorých funkcií rozhrania triedy *Input*. Celkovo je teda bufer reprezentovaný typom `std::deque<int>`.

K zapamätaniu si aktuálnej pozície v buferi sa využíva premenná *cursor_* (na túto premennú sa budeme odkazovať ako na *kurzor*), ktorá udržiava index prvého neprečítaného prvku v buferi (typ tejto premennej je `std::deque<int>::size_type`).

Rozhranie triedy *Input* je navrhnuté tak, aby pripomínalo štandardné rozhranie pre prácu so súbormi. Jeho súčasťou sú nasledujúce metódy (pre úplnosť uvádzame aj návratový typ spolu s parametrami):

`void open(const std::string&)`

Metóda *open()* slúži na otvorenie súboru identifikovaného parametrom tejto metódy. V prípade, že pri zavolaní už bol nejaký súbor otvorený, je tento súbor zatvorený. Metóda zároveň vyprázdni bufer a resetuje pomocné údaje (kurzor a počítadlá riadkov a znakov).

`void close()`

Metóda *close()* zatvorí otvorený súbor. Ak nie je žiadny otvorený, tak metóda neurobí nič. Táto metóda nikdy nemení obsah buferu ani pomocných údajov.

`int get()`

Metóda *get()* tvorí základ práce so vstupným textom. Slúži na čítanie textu znak po znaku. Metóda vracia znak (ako `int`) z buferu na pozícii, na ktorú ukazuje kurzor a zároveň posunie kurzor o jednu pozíciu dopredu. V prípade, že v buferi nie je dostatočný počet znakov, je do buferu načítaný zo vstupného súboru jeden

znak. V prípade, že už žiadny znak nie je dostupný ani vo vstupnom súbore, vráti metóda hodnotu `EOF`.

```
int peek(unsigned int)
std::string peek(unsigned int, unsigned int)
```

Metóda *peek()* slúži na nahliadnutie do vstupného textu za aktuálne čítanú pozíciu (za kurzor). Táto metóda existuje v dvoch podobách, ktoré sa od seba líšia počtom parametrov aj výstupným typom.

Prvá podoba dostáva jeden parameter, ktorý určuje koľko znakov má byť preskočených za aktuálnym kurzorom a vracia prvý znak (znova ako `int`) za preskočenými znakmi. V prípade, že v buferi nie je dostatok znakov, je potrebný počet znakov načítaný zo vstupného súboru. V prípade, že potrebný počet znakov nie je dostupný ani vo vstupnom súbore, vracia metóda hodnotu `EOF`.

Druhá podoba dostáva dva parametre, z ktorých prvý určuje koľko znakov má byť preskočených za aktuálnym kurzorom a druhý určuje koľko znakov má byť vrátených za preskočenými znakmi. Tieto znaky sú vrátené v podobe znakového reťazca (`std::string`). V prípade, že v buferi nie je dostatok znakov na vytvorenie reťazca danej dĺžky, je potrebný počet znakov načítaný zo vstupného súboru. Ak nie je dostatočný počet znakov dostupný ani vo vstupnom súbore, je navrátený kratší reťazec podľa počtu dostupných znakov (eventuálne môže byť vrátený aj prázdny reťazec).

```
void remove(int)
```

Metóda *remove()* slúži na uvoľnenie už spracovaných znakov z buferu. Táto metóda dostáva jeden parameter, ktorý predstavuje počet znakov, ktoré sa majú odstrániť zo začiatku buferu. Pri odstraňovaní jednotlivých znakov sa neberie ohľad na to, či sa nachádzajú pred alebo za aktuálnym kurzorom. V prípade, že sa v buferi nenachádza dostatok znakov, sú potrebné znaky načítané zo vstupného súboru. Ak sa nenachádza dostatočný počet znakov ani vo vstupnom súbore, je zostávajúci počet ignorovaný. Pri odstraňovaní znakov z buferu metóda zároveň automaticky upraví pozíciu kurzora.

```
void reset()
```

Táto metóda nastaví kurzor na prvú pozíciu v buferi. Tým v spolupráci s metódou *remove()* umožňuje v prípade potreby návrat v čítanom texte a prípadné spracovanie iným spôsobom.

```
bool finished() const
```

Metóda *finished()* signalizuje, či ešte existuje nejaký neprečítaný znak. Ak boli zo vstupného súboru načítané všetky dostupné znaky a v buferi sa nenachádza žiadny neprečítaný znak, vracia metóda hodnotu `true`, v opačnom prípade `false`.

```
unsigned int line() const
```

Táto metóda vracia číslo riadku posledného znaku odstráneného z bufferu. Číslovanie začína číslom 1.

```
unsigned int column() const
```

Táto metóda vracia číslo stĺpca prvého znaku v buferi bez ohľadu na to, či sa nachádza pred alebo za kurzorom. Číslovanie začína číslom 1 a je resetované vždy, keď je z buferu odstránený znak reprezentujúci nový riadok (`'\n'`).

```
int read(int = 1)
```

O samotné načítavanie znakov zo vstupného súboru do buferu sa stará metóda

read(). Táto metóda dostáva ako parameter počet znakov, ktoré má vyzdvihnúť z aktuálne čítanej pozície vo vstupnom súbore a vložiť ich do buferu (prednastavená hodnota tohto parametra je 1). Číslo, ktoré metóda vracia, predstavuje počet znakov, ktoré sa v skutočnosti zo vstupného súboru podarilo načítať.

V prípade, že zo vstupného súboru nie je možné čítať (napríklad nie je otvorený žiadny vstupný súbor), vyvolá zavolanie tejto metódy výnimku.

`Input(bool)`
Z podobných dôvodov ako pri doteraz popísaných triedach je konštruktor triedy *Input* privátny, za zmienku však stojí parameter tohto konštruktora. Pri lexikálnej analýze môže byť niekedy výhodné, ak sa pri spracovávaní vstupného textu neberie ohľad na veľkosť jednotlivých znakov (case-sensitivity). Z tohto dôvodu je v triede *Input* možné nastaviť príznak, vďaka ktorému sa bude celý načítavaný text automaticky prevádzať na jeho malú variantu.

Na nastavenie tohto príznaku slúži práve parameter konštruktora - pri inicializácii s hodnotou `false` bude automaticky prevádzaná konverzia znakov vstupného textu na malú variantu, zatiaľ čo pri hodnote `true` bude táto konverzia vynechaná a všetky načítané znaky budú odpovedať ich pôvodnej variante vo vstupnom súbore.

Pre úplné pochopenie triedy *Input* je ešte možné nahliadnuť na stavy, ktorými prechádzajú vstupné znaky pri spracovaní touto triedou. Tieto stavy nie sú v rámci triedy nijako fyzicky implementované a sú skôr dané operáciami, ktoré sa nad danými znakmi prevádzajú. Týmito stavmi sú:

1. Nenačítaný (unread) - znak, ktorý sa nachádza vo vstupnom súbore a ešte nebol načítaný do bufera.

2. Načítaný (loaded) - znak, ktorý bol zo vstupného súboru načítaný do buferu pomocou metódy *peek()*. Tento znak sa spravidla nachádza za kurzorom.

3. Prečítaný (read) - znak, ktorý bol prečítaný pomocou metódy *get()*. Tento znak mohol byť načítaný zo vstupného súboru do buferu pomocou tejto metódy alebo mohol byť načítaný nejakým predchádzajúcim zavolaním metódy *peek()*. Tento znak sa spravidla nachádza pred kurzorom a jeho opätovné vrátenie pomocou metód *get()* a *peek()* nie je možné bez toho, aby bol prevedený späť do stavu loaded pomocou metódy *reset()*.

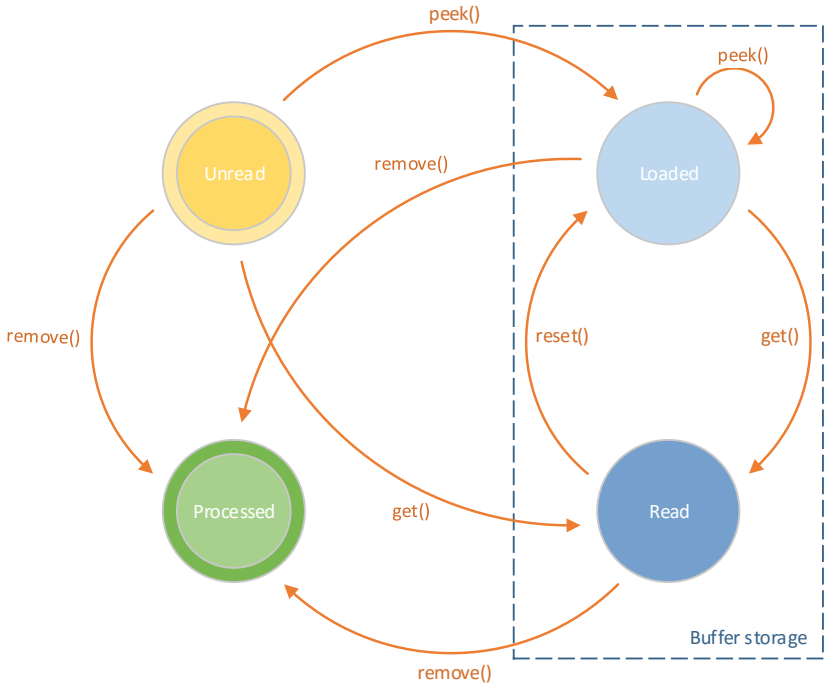
4. Spracovaný (processed) - znak, ktorý už bol z buferu odstránený pomocou metódy *remove()*(znak mohol byť v akomkoľvek z predchádzajúcich stavov). Opätovné vrátenie tohto znaku pomocou metód *peek()* alebo *get()* už nie je možné.

Tieto stavy a operácie zodpovedné za prechod medzi nimi ilustruje obrázok 6.3.

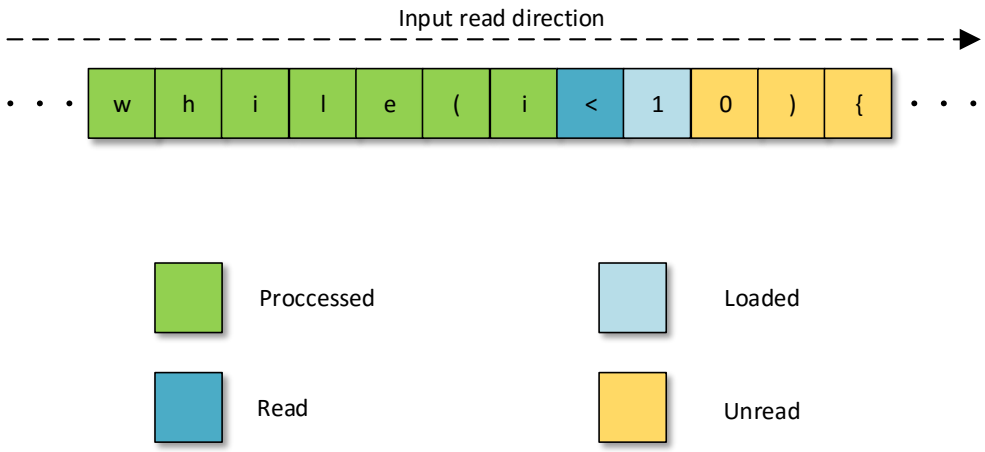
Obrázok 6.4 zobrazuje ako by mohli vyzerať stavy jednotlivých znakov na práve spracovávanej časti vstupného textu.

6.2.6 RingBuffer

Pri syntaktickej analýze je vhodné mať k dispozícii výhľad na niekoľko nasledujúcich terminálov (výhľadové okno), pričom tento výhľad má zvyčajne nejakú



Obr. 6.3: Životný cyklus znaku v triede *Input*.



Obr. 6.4: Ukážka stavu znakov vzhľadom na vstup.

maximálnu veľkosť. Trieda *RingBuffer* predstavuje dátovú štruktúru, ktorá nám umožňuje jednoducho implementovať a efektívne spravovať takéto výhľadové okno. Jej účelom je uchovávať postupnosť nejakých prvkov do pevne stanovenej maximálnej veľkosti a umožniť rýchly prístup k týmto prvkom ako aj ich rýchle pridávanie na koniec postupnosti a odoberanie zo začiatku postupnosti.

RingBuffer je šablónová trieda, kde sa ako jediný šablónový parameter T predpokladá typ ukladaných prvkov. K uloženiu prvkov používa trieda *RingBuf-*

fer dátovú štruktúru `std::vector<T>` zo štandardnej knižnice jazyka *C++*. Táto štruktúra sama o sebe neposkytuje efektívne odoberanie prvkov zo začiatku, pretože je nutné posunúť všetky zvyšné prvky v štruktúre a zároveň neumožňuje vynútenie maximálnej veľkosti. Z tohto dôvodu poskytuje trieda *RingBuffer* nad touto štruktúrou rozhranie, ktoré túto funkcionalitu efektívne poskytuje.

Poznámka 6.7:
K výberu štruktúry `std::vector` ako základu triedy *RingBuffer* je nutné poznamenať, že v štandardnej knižnici existujú dátové štruktúry, ktoré už poskytujú aspoň časť funkcionality, o ktorú sa snaží trieda *RingBuffer*. Príkladom môže byť štruktúra `std::deque`, ktorá umožňuje efektívne odoberanie prvkov zo začiatku alebo štandardné pole (array), ktoré má explicitne danú veľkosť. Žiadna z týchto štruktúr však neposkytuje všetku potrebnú funkcionalitu zároveň. Na druhej strane je vektor ako základná štruktúra dostačujúca, na rozdiel od poľa poskytuje množstvo funkcionality, ktorá môže byť využitá pri prípadnom rozširovaní (napríklad iterátory) a na rozdiel od `std::deque` je použiteľná pri interfacovaní s kódom v jazyku *C*, čo môže byť v budúcnosti tiež využiteľné.

Celá implementácia triedy *RingBuffer* je založená na rozlíšení voľných a zaplnených pozícii v buferi (reprezentovanom pomocou vektora) vopred alokovanej konštantnej veľkosti a o predstave bufferu ako kruhu, teda, že pozícia za poslednou pozíciou v buferi odpovedá prvej pozícii v buferi. Trieda si uchováva informáciu o pozícii prvého prvku (first) a o celkovom počte (count) prvkov aktuálne uložených v buferi. Vďaka tomu je odobratie prvého prvku jednoducho implementované posunutím indexu prvého prvku o jednu pozíciu vopred - čo síce fyzicky hodnotu zo štruktúry neodstráni, ale len zabudneme, že tam nejaká hodnota bola. Podobne je implementované pridanie prvku na koniec - prvok na nasledujúcej voľnej pozícii je prepísaný na novú hodnotu a inkrementovaná je informácia o počte prvkov v štruktúre.

Pri prevádzaní týchto operácií je samozrejme nutné kontrolovať, či ešte je nejaké voľné miesto pre nový prvok, respektíve, či sa nesnažíme odstrániť prvok z prázdnej štruktúry. Kontrola týchto podmienok nám zároveň zaručuje konštantnú maximálnu veľkosť bufera.

Rozhranie triedy *RingBuffer* je navrhnuté tak, aby korešpondovalo s rozhraniami kontajnerov zo štandardnej knižnice jazyka *C++*. Tomu sú prispôsobené aj mená jednotlivých metód a typov, ktoré táto trieda poskytuje. Trieda teda implementuje nasledovné rozhranie:

`size_type`
Trieda *RingBuffer* definuje celočíselný typ `size_type`, ktorý podobne ako v iných štandardných kontajneroch definuje maximálny možný počet prvkov, ktorý môže kontajner držať. Hodnoty tohto typu sa zároveň využívajú ako parametre a návratové hodnoty niektorých metód.

`void push_back(const T&)`
Vloží nový prvok predaný pomocou parametra na prvú voľnú pozíciu v kontajneri. Ak je už kontajner úplne naplnený, vyvolá táto metóda výnimku.

`void pop_front()`
Táto metóda odstráni prvok na prvej pozícii v kontajneri. V prípade, že je kontajner prázdny, vyvolá táto metóda výnimku. V skutočnosti táto metóda žiadny

prvok z kontajnera fyzicky neodstraňuje, ale namiesto toho len inkrementuje index na prvý prvok, a tak by vlastne nikdy výnimku vyvolávať nemusela. Vyvolanie výnimky však odpovedá logike tejto metódy - metóda odstraňuje prvok, a teda, ak nie je možné odstrániť prvok, malo by sa zavolanie tejto metódy považovať za výnimočný stav.

`const T& operator[](size_type)`
Tento operátor slúži na sprístupnenie prvku v kontajneri na pozícii predanej pomocou parametra. Pri získavaní prvku je žiadaná pozícia prepočítaná na pozíciu odpovedajúcu aktuálnemu stavu štruktúry a užívateľ teda žiadny prepočet prevádzať nemusí. Zároveň je pri získavaní prvku kontrolovaná dolná a horná hranica a pri žiadosti o prvok na pozícii, ktorá aktuálne nie je obsadená je vyvolaná výnimka.

`bool empty() const`
Kontroluje, či sa v kontajneri nachádza nejaký prvok. V prípade, že je kontajner prázdny, vracia metóda hodnotu `false`, inak vracia hodnotu `true`.

`size_type size() const`
Vracia aktuálny počet prvkov nachádzajúcich sa v kontajneri.

`void clear()`
Vyprázdni kontajner. V skutočnosti žiadne prvky z kontajnera neodstraňuje, ale len nastavuje príznaky tak, aby sa kontajner javil ako prázdny.

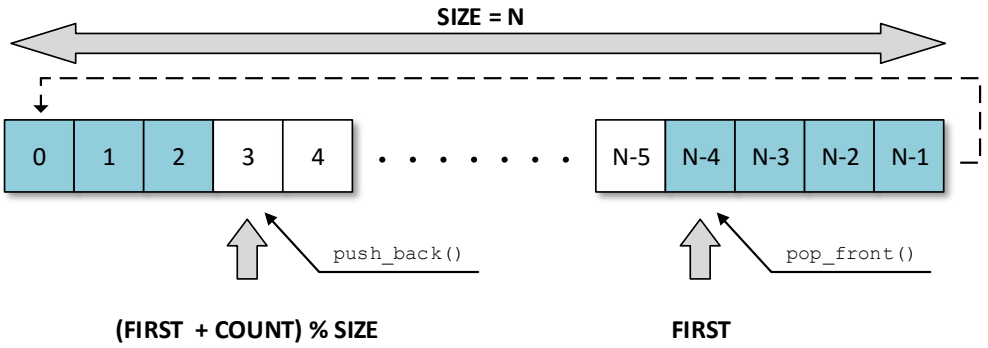
`size_type indexShift(size_type)`
Táto metóda vykonáva prepočet pozície predanej ako parameter na pozíciu v kontajneri, ktorá požadovanej pozícii odpovedá. Metóda pre použitie kontajnera užívateľom nie je potrebná, a preto je označená ako privátna.

`RingBuffer(size_type)`
Aj keď je *RingBuffer* možné použiť s takmer akýmkoľvek typom, jeho primárne určenie v rámci analyzéra je na uchovávanie výhľadového okna. Z tohto dôvodu je konštruktor tejto triedy označený ako privátny. Ako jediný parameter berie konštruktor číslo udávajúce veľkosť vytváraného kontajnera. Z logického hľadiska nemá zmysel vytvárať kontajner o veľkosti menšej ako 1, a preto je v takom prípade vyvolaná výnimka.

Obrázok 6.5 zobrazuje príklad objektu triedy *RingBuffer*. Na obrázku je bufer veľkosti N, obsahujúci 7 prvkov ($COUNT$), ktoré sú zobrazené modrou farbou. Kontajner si uchováva informáciu o pozícii prvého prvku v kontajneri ($FIRST$) a všetky ostatné prvky nasledujú za ním v súvislom bloku (kontajner imituje prepojenie prvej a poslednej pozície v štruktúre). Pri zavolaní metódy *pop_front()* sa inkrementuje $FIRST$ modulo N a dekrementuje $COUNT$. Pri zavolaní metódy *push_back()* sa vkladá prvok na pozíciu $FIRST + COUNT$ modulo N, čo odpovedá pozícii za posledným prvkom. Pri volaní oboch metód sa samozrejme kontroluje dodržiavanie invariantov a to, či v kontajneri existuje nejaký prvok, respektíve, či nie je kontajner už naplnený.

6.2.7 LexicalAnalyzer

Trieda *LexicalAnalyzer* predstavuje základ lexikálneho analyzátora používaného pri analýze vstupného súboru. Jedná sa o abstraktnú šablónovú triedu, ktorá



Obr. 6.5: Štruktúra RingBufferu.

definuje rozhranie, dátové typy a dátové štruktúry pre všetky vygenerované lexikálne analyzátory. Ako hodnota šablónového parametra sa predpokladá dátový typ *Terminal* odpovedajúceho (syntaktického) analyzátora.

Pre prácu akéhokoľvek lexikálneho analyzátora je nutné byť schopný pracovať so vstupným textom. K tomu je v triede *LexicalAnalyzer* definovaná inštancia už predstavenej triedy *Input*. S touto inštanciou zároveň úzko súvisia nasledujúce metódy triedy *LexicalAnalyzer*, ktoré s ňou umožňujú pracovať:

`void openInput(const std::string&)`
Táto metóda v podstate reštartuje prácu lexikálneho analyzára. Pri jej zavolaní je vyčistený kontajner s tokenmi a je otvorený vstupný súbor špecifikovaný pomocou parametra metódy, ktorý bude použitý k lexikálnej analýze. Táto metóda je verejná (public) a využíva sa syntaktickým analyzátorom k započatiu lexikálnej analýzy.

`void closeInput()`
Táto metóda, ako už názov napovedá, ukončuje prácu so vstupom. Pri jej zavolaní dôjde k zavretiu súboru otvoreného pomocou metódy *openInput()*, avšak kontajner s tokenmi je zachovaný v pôvodnom stave. Rovnako ako predchádzajúca metóda je táto metóda verejná a používa sa syntaktickým analyzátorom k ukončeniu práce so vstupom.

`Input& input()`
Táto metóda vracia referenciu na inštanciu triedy *Input* definovanej v rámci triedy *LexicalAnalyzer*. Metóda je označená ako chránená (protected), pretože jej účelom je sprístupniť túto inštanciu triedam, ktoré implementujú konkrétny lexikálny analyzátor a sú odvodené od triedy *LexicalAnalyzer* (viď ďalej triedu *Lexer*). Zároveň však chceme zabrániť prístupu k tejto inštancii mimo hierarchiu tried implementujúcich konkrétny lexikálny analyzátor. Tento fakt je aj dôvodom, prečo museli byť samostatne definované metódy *openInput()* a *closeInput()*, namiesto zavolania odpovedajúcej metódy na inštancii triedy *Input*.

Ďalšou nepostrádateľnou súčasťou lexikálneho analyzátora je systém uchovávania a poskytovania tokenov získaných zo vstupného textu. Táto funkcionalita je založená na použití už predstavenej triedy *RingBuffer*, ktorej inštancia definovaná v rámci triedy *LexicalAnalyzer* predstavuje výhľadové okno. V súvislosti s výhľadovým oknom a množinou identifikátorov terminálov definuje trieda *Le-*

xicalAnalyzer niekoľko typov:

`typedef Token<T> Token`
Predstavuje token špecializovaný pre konkrétnu množinu identifikátorov terminálov definovaných parametrom *T*.

`typdef RingBuffer<Token> TokenBuffer`
Predstavuje kontajner pre špecializovaný typ tokenov.

`typedef typename TokenBuffer::size_type size_type`
Odpovedá typu *size_ type* z triedy *RingBuffer* špecializovanej typom *Token*. Tento typ sa používa ako typ indexu pri prístupe do výhľadového okna.

Prvé dva definované typy sa využívajú v internej implementácii lexikálneho analyzátora, a preto sú označené ako chránené, tretí typ však slúži pri prístupe do výhľadového okna, a preto je označený ako verejný, aby bol prístupný triede implementujúcej syntaktickú analýzu.

S výhľadovým oknom ďalej súvisia nasledujúce metódy:

`virtual T operator[](size_type)`
Táto metóda slúži k nahliadnutiu do výhľadového okna. Pri zavolaní je vrátený identifikátor terminálu na pozícii reprezentovanej parametrom metódy. Implementácia tejto metódy je závislá na konkrétnych hodnotách identifikátorov terminálov, preto je označená ako čisto virtuálna a jej implementácia je ponechaná na triedu reprezentujúcu konkrétny plne špecializovaný lexikálny analyzátor.

`bool matchTerminal(const T&, std::vector<Attribute>&)`
Úlohou tejto metódy je spracovanie terminálov vo výhľadovom okne. Metóda overí, či sa na prvej pozícii výhľadového okna nachádza token s terminálnym identifikátorom odpovedajúcim hodnote prvého parametra. V prípade, že sa identifikátory líšia, vracia metóda hodnotu `false`. V prípade, že sa identifikátory zhodujú, je token na prvej pozícii výhľadového okna odstránený, jeho atribút je vložený do kontajnera predaného pomocou druhého parametra a je vrátená hodnota `true`. V prípade, že sa vo výhľadovom okne nenachádza žiadny token (lebo napríklad ešte nebol žiadny získaný alebo už boli všetky získané tokeny spracované), sa metóda najprv pokúsi nejaký token získať zo vstupného textu.

O načítavanie nových tokenov do výhľadového okna sa starajú dve metódy:

`virtual void fetchToken() = 0`
Metóda sa pokúsi získať zo vstupného textu nový token a vložiť ho do výhľadového okna na prvú voľnú pozíciu. Keďže každý lexikálny analyzátor rozpoznáva špecifické tokeny odpovedajúce jeho vstupnej gramatike, je implementácia tejto metódy ponechaná až do implementácie konkrétneho lexikálneho analyzátora.

`void fillLookahead()`
Táto metóda slúži na naplnenie výhľadového okna. Využíva pritom metódu *fetchToken()*, ktorá je volaná pokiaľ nie je výhľadové okno naplnené alebo už nie je možné zo vstupného textu získať žiadny token (napr. preto, že bol prečítaný celý vstupný text).

Použitie metód *fetchToken()* a *fillLookahead()* pri implementácii metód *operator[]()* a *matchTerminal()* výrazne uľahčuje prácu s lexikálnym analyzátorom. Užívateľ (syntaktický analyzátor) proste požiada o terminál na istej pozícii v rámci výhľadu, prípadne požiada o spracovanie prvého terminálu a má istotu, že jeho

požiadavka bude vybavená (ak je to možné) bez toho, aby musel špecificky žiadať o naplnenie výhľadu.

Z pohľadu syntaktického analyzátora sú metódy *operator[]()* a *matchTerminal()* dostačujúce k jeho efektívnej práci. Z pohľadu lexikálneho analyzátora je však nutné mať nad výhľadovým oknom väčšiu kontrolu. Preto trieda *LexicalAnalyzer* obsahuje ešte chránenú metódu *tokens()*, ktorá vracia referenciu na kontajner udržujúci výhľadové okno (`TokenBuffer&`) a tým ho sprístupňuje aj triede reprezentujúcej konkrétny lexikálny analyzátor.

Okrem doteraz spomínaných členov trieda *LexicalAnalyzer* ďalej definuje typ určený pre uchovanie tvarov kľúčových slov a ich asociáciu s patričným identifikátorom terminálu. Týmto typom je:

```
typedef std::map<std::string, T> Keywords
```

Jedná sa o mapu, ktorá udržuje dvojice (tvar, identifikátor). Typ `std::map` bol vybraný vďaka jeho vlastnostiam, hlavne kvôli tomu, že prvky v tomto kontajneri sú zoradené podľa kľúča (v tomto prípade podľa tvaru kľúčového slova), čo je využité pri hľadaní kľúčových slov vo vstupnom texte.

Spolu s týmto typom definuje trieda *LexicalAnalyzer* ešte aj chránenú metódu keywords(), ktorá vracia referenciu na kontajner typu *(*Keywords). Napriek tomu, že nie každý lexikálny analyzátor podporuje kľúčové slová, bola táto funkcionalita pridaná už do triedy *LexicalAnalyzer* (namiesto do triedy implementujúcej konkrétny lexikálny analyzátor s kľúčovými slovami), pretože to dobre odpovedá náhľadu na lexikálny analyzér ako celok a rovnako to tvorí dobrý základ pre budúce rozšírenia lexikálneho analyzátora.

Na rozdiel od doteraz prezentovaných tried má trieda *LexicalAnalyzer* chránený konštruktor. Dôvodom k tomuto kroku je fakt, že trieda je určená k dedeniu, a preto musí byť jej konštruktor prístupný derivovaným triedam. Konštruktor tejto triedy dostáva dva parametere, z ktorých prvý predstavuje veľkosť výhľadového okna a druhý signalizuje, či sa má pri analýze vstupného textu rozlišovať medzi malou a veľkou formou čítaných znakov.

Konkrétny lexikálny analyzátor musí mať možnosť vytvárať inštancie tried *Attribute* a *Token*, ktorých konštruktory sú privátne. Keďže je však konkrétny lexikálny analyzátor vždy dynamicky generovaný pre každú gramatiku zvlášť, nie je možné v týchto triedach označiť každú triedu reprezentujúcu konkrétny lexikálny analyzátor ako priateľskú triedu (`friend class`). Z tohto dôvodu poskytuje trieda *LexicalAnalyzer* ešte nasledujúce dve chránené metódy:

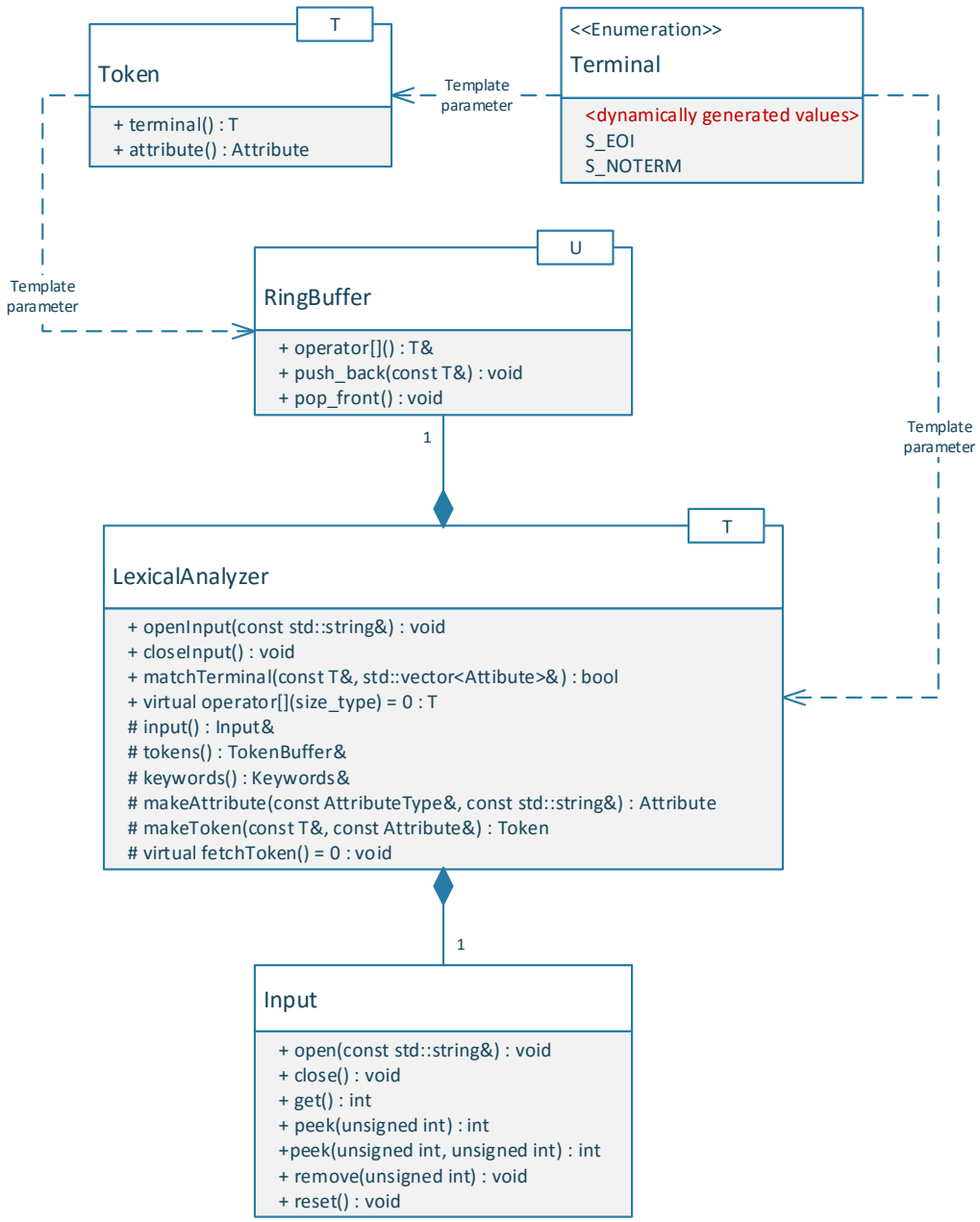
```
Attribute makeAttribute(const AttributeType&, const std::string&)
```
Táto metóda na základe poskytnutých parametrov vytvorí a vráti inštanciu triedy *Attribute*.

```
Token makeToken(const T&, const Attribute&)
```
Táto metóda na základe poskytnutých parametrov vytvorí a vráti inštanciu typu *Token*.

Vďaka tomu, že sú tieto metódy označené ako chránené, je ich použitie obmedzené len na triedy derivované od triedy *LexicalAnalyzer* a tým sa stále zachováva obmedzenie vytvárania inštancií tried *Attribute* a *Token* mimo vybrané priateľské triedy.

Obrázok 6.6 ilustruje vzťahy medzi triedami *Input*, *RingBuffer*, *Token*, *Ter-*

minal a *LexicalAnalyzer*.



Obr. 6.6: UML diagram popisujúci vzťah typu Terminal a tried Token, RingBuffer, LexicalAnalyzer a Input.

6.2.8 Lexer

Trieda *Lexer* implementuje konkrétny lexikálny analyzátor pre danú gramatiku. Jedná sa o dynamicky generovanú triedu, ktorá je pomocou dedičnosti odvodená od triedy *LexicalAnalyzer* a špecializovaná typom *Terminal*, ktorý odpovedá rovnakej gramatike. Keďže všetky potrebné štruktúry sú definované v triede *LexicalAnalyzer*, ostáva už len implementovať virtuálnu metódu *fetchToken()*

(na vyhľadávanie tokenov vo vstupnom texte) a virtuálny operátor *operator[]()* (na sprístupnenie výhľadového okna).

`Terminal operator[](size_type) override`
Vďaka špecializácii triedy pomocou typu *Terminal* je možné implementovať tento virtuálny operátor. Ako už bolo spomenuté pri popise triedy *LexicalAnalyzer* úlohou tohto operátora je poskytovať prístup do výhľadového okna triede implementujúcej syntaktický analyzátor. Operátor teda jednoducho vracia identifikátor terminálu vo výhľadovom okne na pozícii reprezentovanej parametrom. Táto pozícia je samozrejme kontrolovaná oproti veľkosti výhľadového okna a v prípade, že nespadá do predpokladaných medzí, je vyvolaná výnimka. Zároveň je však nutné riešiť prípad, že sa syntaktický analyzátor pomocou tohto operátora dožaduje identifikátora na pozícii v rámci výhľadového okna, avšak vo vstupnom súbore už nie je dosť terminálov, aby bolo možné vrátiť ten, ktorý by danej pozícii odpovedal. Preto je v type *Terminal* definovaná hodnota `Terminal::S_NOTERM`, ktorá slúži ako návratová hodnota v tomto prípade. Keďže však návratový typ nebolo možné presne špecifikovať už v triede *LexicalAnalyzer*, musela byť implementácia tohto operátora ponechaná až na triedu *Lexer*.

Poznámka 6.8:
Problém operátora bolo možné riešiť aj inými spôsobmi, napríklad jeho nahradením metódou, ktorá by okrem vrátenia hodnoty terminálneho identifikátora ešte nastavovala výstupný parameter indikujúci validitu návratovej hodnoty. Takáto metóda by bola implementovateľná už v triede *LexicalAnalyzer*, práca s ňou by však bola nemotornejšia a výsledný vygenerovaný kód by len zneprehľadňovala. Naopak, zavedenie špeciálnej hodnoty `Terminal::S_NOTERM`, ktorá neodpovedá žiadnemu terminálnemu identifikátoru, dáva z funkčného hľadiska zmysel a zároveň zjednodušuje kód.

`void fetchToken() override`
Metóda *fetchToken()* implementuje získanie nasledujúceho tokenu v rámci vstupného textu a jeho následné vloženie do výhľadového okna. Dôvodom, prečo je táto funkcionalita implementovaná až v triede *Lexer*, je jednoducho fakt, že každý lexikálny analyzátor odpovedá inej gramatike, a teda pracuje s inou množinou terminálov a môže využívať rôzne techniky ich vyhľadávania vo vstupnom texte.

Implementácia tejto metódy je dynamicky generovaná a spravidla ju tvoria metódy implementujúce vyhľadávanie jednotlivých druhov terminálov (čísel, kľúčových slov, symbolov, identifikátorov, ...). Tieto metódy sú prítomné v závislosti na definovanej gramatike. Napríklad lexikálny analyzátor pre gramatiku, ktorá nepoužíva čísla, nebude obsahovať metódu, ktorá čísla vo vstupnom texte vyhľadáva.

V prípade, že sa metóde *fetchToken()* nepodarí zo vstupného textu získať žiadny token aj napriek tomu, že ešte nebol spracovaný celý vstupný súbor, je vyvolaná výnimka.

O vyhľadávanie jednotlivých tokenov vo vstupnom texte sa starajú nasledujúce metódy:

`void skipWhitespace()`
Táto metóda zabezpečuje odstránenie všetkých bielych znakov (whitespace) zo začiatku nespracovanej časti vstupného textu. Metóda je vygenerovaná v prípade,

že sa majú ignorovať biele miesta (záznam `ignorewhitespace` v konfiguračnom súbore).

Návratový typ je `void`, pretože metóda neslúži na vyhľadávanie žiadnych tokenov vo vstupnom texte, ale len odstraňuje prebytočné biele miesta, a preto nepotrebuje signalizovať svoj úspech či neúspech.

`bool parseWhitespace()`
Táto je vygenerovaná v prípade, že sa biele miesta nemajú preskakovať a v prípade, že na začiatku nespracovanej časti vstupného textu narazí na nejaké biele miesto, vytvorí odpovedajúci token, ktorý zaradí na prvú voľnú pozíciu vo výhľadovom okne a vráti hodnotu `true`. V opačnom prípade len vráti hodnotu `false`.

`void skipComments()`
Metóda *skipComments()* je vygenerovaná v prípade, že gramatika definuje nejaký druh komentárov (položky `commentstart` a `commentend` alebo `singlecomment` v konfiguračnom súbore). V prípade, že sa na začiatku nespracovanej časti vstupného textu nachádza validný komentár, táto metóda ho odstráni. Ak je nájdený nevalidný komentár (viacriadkový komentár, ktorý nie je ukončený do konca súboru), vyvolá táto metóda výnimku. Podobne ako metóda *skipWhitespace* je návratový typ tejto metódy `void`.

`bool parseNumber()`
Vygenerovanie metódy *parseNumber()* záleží na tom, či daná gramatika definuje terminál pre celé resp. desatinné čísla (položka `numterminal` resp. `realterminal` v konfiguračnom súbore). Implementácia metódy je prispôsobená typu čísla, ktoré má na vstupe hľadať. V prípade, že je nejaké číslo nájdené, je vytvorený príslušný token, ktorý je zaradený do výhľadového okna a je vrátená hodnota `true`. Ak nie je nájdené žiadne číslo, je vrátená hodnota `false`.

`bool parseKeywordOrIdentifier()`
Úlohou tejto metódy je rozpoznávať v texte kľúčové slová a identifikátory. Jej vygenerovanie a implementácia závisí na položke `identifierterminal` a na počte definovaných kľúčových slov v konfiguračnom súbore. Metóda porovnáva najdlhší prefix nespracovanej časti vstupného textu začínajúci písmenom, za ktorým nasledujú alfanumerické znaky alebo znak `'_'` s definovanými tvarmi kľúčových slov. V prípade, že je nájdená zhoda, je vytvorený token pre patričné kľúčové slovo, v prípade, že zhoda nie je nájdená, je vytvorený token pre identifikátor; daný token je zaradený do výhľadového okna a je vrátená hodnota `true`. Ak prefix nezačína písmenom, je vrátená hodnota `false`.

`bool parseChar()`
Metóda *parseChar()* slúži na rozpoznávanie znakových konštánt. Jej vygenerovanie je závislé na položke `charterminal` v konfiguračnom súbore a implementácia je závislá na položke `chardelim` v konfiguračnom súbore. Ak nespracovaná časť vstupného textu začína znakom, ktorý odpovedá oddeľovaču znakových konštánt (`chardelim`), za ktorým nasleduje jeden znak (prípadne dva, ak je prvý znak `'\'`) a potom znova oddeľovač znakových konštánt, je vytvorený token pre odpovedajúcu znakovú konštantu a vrátená hodnota `true`. V opačnom prípade je vrátená hodnota `false`.

`bool parseString()`
Metóda *parseString()* slúži na rozpoznávanie znakových reťazcov a jej vygenero-

vanie je závislé na položke `stringterminal` v konfiguračnom súbore a implementácia je závislá na položke `stringdelim` v konfiguračnom súbore. Spôsob vyhľadávania je podobný ako v metóde *parseChar()*; ak nespracovaná časť vstupného textu začína oddeľovačom znakových reťazcov (`stringdelim`) nasledovaným reťazcom znakov ukončeným oddeľovačom znakových reťazcov, pred ktorým sa nenachádza znak `'\'`, je vytvorený príslušný token, ktorý je zaradený do výhľadového okna a je vrátená hodnota `true`. V opačnom prípade je vrátená hodnota `false`.

`bool parseSymbol()`
Vygenerovanie a implementácia tejto metódy je závislá na symboloch definovaných v konfiguračnom súbore. Metóda sa snaží nájsť čo najdlhší prefix nespracovanej časti vstupného textu, ktorá celá odpovedá niektorému zo symbolov. K tomu je použitá (pomerne členitá) rozhodovacia štruktúra zložená z `if` a `case` blokov. V prípade, že sa podarí nájsť prefix odpovedajúci nejakému symbolu a je isté, že žiadny iný (dlhší) symbol už nebude odpovedať, je vytvorený odpovedajúci token, ktorý je následne zaradený do výhľadového okna a je vrátená hodnota `true`. V opačnom prípade je vrátená hodnota `false`.

`bool parseEOI()`
Táto metóda slúži na rozpoznanie konca vstupného textu a je vždy prítomná. Ak vo vstupnom texte už nie je žiadny nespracovaný znak okrem znaku konca súboru (`EOF`), je vytvorený token odpovedajúci koncu vstupného textu, ktorý je zaradený do výhľadového okna a je vrátená hodnota `true`, inak je vrátená hodnota `false`.

Okrem týchto metód obsahuje trieda *Lexer* ešte niekoľko metód, ktoré definujú konštanty používané pri lexikálnej analýze. Cieľom týchto metód je zabezpečiť jednoduchšiu čitateľnosť a údržbu kódu. Príkladom takejto metódy je napríklad:

`char decimalPoint() const`
Táto metóda definuje znak predstavujúci oddeľovač desatinnej časti v desatinnom čísle.

Použitím vyššie spomenutých metód implementuje funkcia *fetchToken()* vyhľadávanie tokenov vo vstupnom texte. Metóda začína hľadať tokeny na prvej nespracovanej pozícii a každé zavolanie tejto metódy nájde nanajvýš jeden token, pričom vstupný text, ktorý tvoril nájdený token, sa bude pri ďalšom spustení považovať za už spracovaný. Aj keď metóda *fetchToken()* žiadny token nevracia, budeme nájdenie, vytvorenie a zaradenie tokenu do výhľadového okna označovať ako vrátenie tokenu. Nasledujúci postup predstavuje stratégiu použitú pri hľadaní tokenov a pri každom bode sa predpokladá, že daná funkcionalita je požadovaná:

1. Ak bol spracovaný celý vstupný text vrátane znaku pre koniec súboru (`EOF`), tak skonč.
2. (*skipWhitespace() / parseWhitespace()*)
   (a) Ak sa majú ignorovať biele znaky, tak preskakuj znaky, kým nenarazíš na iný ako biely znak, inak,
   (b) ak sa nemajú preskakovať biele znaky a nasleduje biely znak, vráť odpovedajúci token.

3. (*skipComments()*)
   Ak sú definované komentáre, tak pokiaľ

   (a) nasleduje oddeľovač riadkového komentára, preskakuj všetky znaky, kým nenarazíš na koniec riadku alebo súboru,

   (b) nasleduje počiatočný oddeľovač viacriadkového komentára, tak preskakuj všetky znaky, kým nenarazíš na koncový oddeľovač viacriadkového komentára (v prípade, že na koncový oddeľovač nenarazíš do konca súboru, vyvolaj výnimku).

4. (*parseNumber()*)
   Ak nasleduje číslica a za ňou prípadné ďalšie číslice, vráť token odpovedajúci celému číslu.

   V prípade, že sú povolené desatinné čísla, je možné, aby sa v nájdenom čísle nachádzal jeden predstavujúci oddeľovač desatinných miest a v tom prípade je vrátený token odpovedajúci desatinnému číslu. Oddeľovač desatinných miest sa môže nachádzať aj na začiatku (príp. konci) čísla a v tom prípade je vrátený token odpovedajúci desatinnému číslu bez celej (príp. desatinnej) časti.

   V prípade, že je v konfiguračnom súbore povolený vedecký zápis alebo numerické sufixy (položky `allowenotation` a `numericsuffixes`), môže byť súčasťou čísla aj preddefinovaný sufix. Pre číslo, ktoré takýto sufix obsahuje, bude vždy vygenerovaný terminál odpovedajúci desatinnému číslu (aj keď žiadnu desatinnú časť neobsahuje).

5. (*parseKeywordOrIdentifier()*)
   Ak nasleduje písmeno alebo znak `'_'` a za ním prípadne ďalšie alfanumerické znaky alebo znak `'_'`, porovnaj získaný reťazec s tvarmi kľúčových slov. V prípade zhody vráť token pre odpovedajúce kľúčové slovo. Ak reťazec neodpovedá žiadnemu kľúčovému slovu, vráť token identifikátoru.

6. (*parseChar()*)
   Ak nasleduje oddeľovač znakových konštánt, za ním znak (prípadne ďalšie dva znaky, ak je prvý znak `'\'`) a potom znova oddeľovač znakových konštánt, vráť token odpovedajúci znakovej konštante.

7. (*parseString()*)
   Ak nasleduje oddeľovač znakových reťazcov a za ním ľubovoľné množstvo znakov ukončené znova oddeľovačom znakových reťazcov (pred ktorým sa nenachádza znak `'\'`), vráť token odpovedajúci znakovému reťazcu.

8. (*(parseSymbol()*)
   Ak nenastala žiadna z predchádzajúcich možností, je prehľadaný strom symbolov a v prípade zhody je vrátený token odpovedajúci príslušnému symbolu.

9. (*(parseEOI()*)
   Ak už bol spracovaný celý vstupný text a ostáva len symbol pre koniec súboru, je vrátený token odpovedajúci terminálu pre koniec súboru.

10. Ak nenastala žiadna z vyššie uvedených možností, je vyvolaná výnimka.

Poznámka 6.9:
Pretože sa táto stratégia snaží byť čo najvšeobecnejšia, nemusí vždy vyhovovať konkrétnym požiadavkám užívateľa. V tom prípade je nutné ručne prispôsobiť implementáciu tejto stratégie.

Jedným z miest, kde by táto stratégia nemusela vyhovovať, by mohol byť napríklad nejaký špeciálny formát desatinných čísel, ktorý by neodpovedal možnostiam poskytovaných generátorom.

Posledná vec, ktorú je potrebné pri triede *Lexer* spomenúť, je jej konštruktor. Konštruktor berie dva parametre, ktoré predáva konštruktoru triedy *LexicalAnalyzer*, od ktorej je trieda *Lexer* odvodená.

Prvý parameter nastavuje veľkosť použitého výhľadového okna. Druhý parameter určuje, či sa pri spravaní vstupného textu rozlišuje medzi veľkými a malými variantami jednotlivých znakov.

V tele konštruktora je navyše inicializovaná množina kľúčových slov vložením dvojíc predstavujúcich tvar kľúčového slova a odpovedajúci identifikátor terminálu do mapy kľúčových slov.

6.2.9 Analyzer

Trieda *Analyzer* predstavuje dynamicky generovanú implementáciu samotného rekurzívne zostupného analyzátora, ktorá obsahuje metódy slúžiace k analýze výstupu lexikáneho analyzátora. Jednotlivé metódy je možné rozdeliť do niekoľkých skupín v závislosti na ich použití:

Metódy verejného rozhrania

`void parse(const std::string&, std::ostream& = std::cout)`
Metóda *parse* slúži k započatiu analýzy vstupného textu. Prvý parameter tejto metódy určuje cestu k súboru, ktorý daný vstupný text obsahuje.

Druhý parameter slúži k definícii prúdu, ktorý bude v rámci analýzy použitý k zaznamenávaniu výstupu (napr. k výpisu výstupných terminálov). Tento parameter očakáva referenciu na typ `std::ostream`, čiže k zaznamenaniu výstupu môže byť použitý napríklad klasický súbor (`std::ofstream`), štandardný výstup (`std::cout`) alebo buffer (`std::ostringstream`). Hodnota tohto parametra je prednastavená na štandardný výstup, a preto je možné ju ignorovať v prípade, že užívateľ nepotrebuje alebo nechce definovať žiadny výstup. Avšak v prípade, že užívateľ chce predefinovať túto hodnotu, je nutné, aby zabezpečil, že do ním definovaného prúdu je možné zapisovať (táto vlastnosť je kontrolovaná pomocou metódy `std::ostream::good`). To napríklad znamená, že ak užívateľ chce presmerovať výstup do súboru, musí zaručiť, že predaný súbor je otvorený pre zápis. Rovnako musí po ukončení analýzy zabezpečiť, že bude daný súbor korektne zatvorený.

Samotným spustením metódy *parse* dôjde k nastaveniu výstupného prúdu a kontrole jeho validity. V prípade, že výstupný prúd nie je pripravený, je vrátená výnimka. Následne je vyčistená štruktúra, ktorá slúži k udržiavaniu nespracovaných atribútov (pretože metódu *parse* je možné spúšťať opakovane, je možné, že spomínaná štruktúra obsahuje artefakty z predchádzajúcich analýz). Následne

je cesta k súboru obsahujúcemu vstupný text predaná inštancii triedy *Lexer* k lexikálnej analýze (pomocou metódy `Lexer::openInput`) a zavolaná metóda slúžiaca k rozvoju štartovacieho neterminálu. Po ukončení rozvoja je súbor so vstupným textom zatvorený pomocou metódy `Lexer::closeInput`.

Metóda *parse* nevracia žiadnu hodnotu. V prípade neúspechu je vyvolaná výnimka, zatiaľ čo v prípade úspechu metóda jednoducho skončí.

`void setLogFile(const std::string&)`

Metóda *setLogFile* slúži na nastavenie súboru, do ktorého sa budú ukladať stopovacie záznamy o priebehu analýzy. Tento súbor je definovaný cestou predanou pomocou parametra metódy.

Metóda je prístupná len v prípade, že je pre daný analyzátor povolené stopovanie (viď triedu *Log*). Ak je táto metóda prístupná, musí byť vždy použitá pred prvým zavolaním metódy *parse*, inak bude vyvolaná výnimka.

Ak už mal log nastavenú nejakú masku (viď nižšie), bude táto maska využívaná aj naďalej.

`void setLogMask(int)`

Táto metóda slúži na nastavenie masky logu. Maska logu definuje, aké typy záznamov majú byť ukladané do súboru s logom. Pred zavolaním tejto metódy musí byť nastavený logovací súbor pomocou metódy *setLogFile*, inak nemá nastavenie masky žiadny efekt.

`int getLogMask() const`

Táto metóda slúži na získanie aktuálnej masky logu. Použitie tejto metódy je vhodné, ak chce užívateľ napríklad len pozmeniť aktuálnu masku a nie ju úplne predefinovať. Podobne ako pri metóde *setLogMask* musí byť pred použitím tejto metódy nastavený logovací súbor pomocou metódy *setLogFile*, v opačnom prípade vráti metóda vždy hodnotu 0.

Metódy práce s výhľadom

`void matchTerminal(const Terminal&)`

Metóda *matchTerminal* slúži k overeniu, že sa na prvej pozícii vo výhľadovom okne nachádza token s terminálnym identifikátorom odpovedajúcim identifikátoru, ktorý bol metóde predaný parametrom. Toto overenie je prevedené metódou `LexicalAnalyzer::matchTerminal`, ktorej je zároveň predaná referencia na štruktúru, do ktorej bude v prípade zhody uložený atribút spracovaného tokenu. Spracovaný token je zároveň odstránený z výhľadového okna.

Keďže sa táto metóda používa len ak má byť spracovaný token z výhľadového okna, je v prípade nezhody identifikátorov vyvolaná výnimka, ktorá signalizuje problém v priebehu analýzy.

`bool compareView(Lexer::size_type, const Terminal&)`

Metóda *compareView* podobne ako metóda *matchTerminal* porovnáva terminálny identifikátor tokenu s identifikátorom predaným pomocou parametra. Jej úlohou je však len signalizovať, či sa dané identifikátory zhodujú a v prípade zhody alebo nezhody už s tokenom neprevádza žiadne operácie.

Druhý parameter tejto metódy predstavuje index na pozíciu vo výhľadovom okne, na ktorej sa má nachádzať token určený k porovnaniu s terminálnym identifikátorom. Terminálny identifikátor tohto tokenu je získaný operátorom `Lexer::operator[]`. Ten v prípade, že je index v rozsahu výhľadového okna,

vždy vráti nejaký terminálny identifikátor (v prípade, že sa na danej pozícii už nenachádza žiadny token, je vrátený identifikátor `Terminal::S_NOTERM`).

Implementácia metódy *compareView* je natoľko jednoduchá, že by ani nepotrebovala byť definovaná v samostatnej metóde - jej implementáciu tvorí v podstate jeden riadok:

```
return lexer_[idx] == terminal;
```

Dôvod, prečo je táto funkcionalita definovaná v samostatnej metóde, je čisto pragmatický. Porovnávanie identifikátorov vo výhľadovom okne je v rámci analyzátora veľmi častá operácia a akékoľvek zmeny alebo vylepšenia tejto funkcionality by tak vyžadovali zásah na mnohých miestach v rámci implementácie analyzátora.

Pomocné metódy

`std::vector<Attribute>& attributes()`
Metóda *attributes* vracia referenciu na štruktúru obsahujúcu atribúty tokenov, ktoré prešli analýzou. Táto metóda je určená k použitiu len v rámci implementácie analyzátora, a preto je implementovaná ako privátna. Jej použitie je taktiež vhodné pre implementáciu sémantických akcií v prípade nutnosti prístupu k atribútom. Nakoľko referencia na štruktúru, vrátená touto metódou, nie je konštantná, je možné meniť obsah štruktúry, a teda je nutné, aby užívateľ s touto štruktúrou narábal opatrne.

`std::ostream& out()`
Metóda *out* sprístupňuje referenciu na výstupný prúd definovaný pri volaní metódy *parse*. Táto funkcia je využívaná len v rámci implementácie analyzátora, a preto je implementovaná ako privátna. Príkladom využitia tejto metódy je implementácia metód pre výstupné terminály, avšak užívateľ ju môže využiť aj pri implementácii sémantických akcií.

`std::string terminalToString(const Terminal&)`
Metóda *terminalToString* vytvára reťazcovú reprezentáciu terminálneho identifikátora predaného parametrom metódy. Táto metóda je závislá na aktuálnej definícii vymenovaného typu *Terminal*, ktorý je definovaný len pre danú triedu predstavujúcu analyzátor, a preto nie je použiteľná vo verejnom rozhraní a je definovaná ako privátna. Metóda sa preto používa len pri vytváraní správ v rámci triedy predstavujúcej analyzátor (napríklad pri vytváraní logovacích správ).

Metódy implementujúce rozvoj neterminálov
Táto skupina metód je dynamicky generovaná podľa nastavení z konfiguračného súboru. Obsahuje metódy implementujúce prechod stromami pravidiel jednotlivých neterminálov. Pre každý neterminál je vygenerovaná práve jedna metóda. Implementácia týchto metód odpovedá štruktúre popísanej v sekcii 4.4.

Každá z týchto metód má nasledujúcu signatúru:

`void nonterminalMethod();`
Teda, všetky metódy implementujúce rozvoj neterminálov majú návratový typ `void` a neakceptujú žiadne argumenty. Názvy týchto metód sú odvodené od názvov neterminálov, ktorých rozvoj implementujú. Spravidla začínajú prefixom `nont_`, za ktorým nasleduje názov samotného neterminálu, napríklad, metóda implementujúca rozvoj neterminálu S by bola deklarovaná následovne:

```
void nont_S();
```

Metódy implementujúce výstupné terminály

Táto skupina metód je rovnako generovaná podľa nastavení v konfiguračnom súbore a implementuje výpis výstupných terminálov do výstupného prúdu definovaného pri zavolaní metódy *parse*. Pre každý výstupný terminál je vygenerovaná práve jedna metóda.

Každá z týchto metód má nasledujúcu signatúru:

```
void outterminalMethod();
```

Teda, všetky metódy implementujúce výpis výstupných terminálov majú návratový typ *void* a neakceptujú žiadne argumenty. Názvy týchto metód sú odvodené od názvov výstupných terminálov, ktorých výpis implementujú. Spravidla začínajú prefixom `out_`, za ktorým nasleduje názov samotného výstupného terminálu, napríklad, metóda implementujúca výpis výstupného terminálu *outplus* by bola deklarovaná nasledovne:

```
void out_outplus();
```

Užívateľsky definované metódy

Podobne ako predchádzajúce dve skupiny metód je aj táto skupina dynamicky generovaná - na rozdiel od nich však implementáciu týchto metód takmer úplne určuje užívateľ a to zadaním zdrojového kódu jednotlivých metód v konfiguračnom súbore (viď sekciu 5.2). Túto skupinu metód je navyše možné rozdeliť na dve poskupiny.

Prvú podskupinu tvoria metódy implementujúce jednotlivé sémantické akcie. Tieto metódy sú volané v metódach implementujúcich rozvoj jednotlivých neterminálov. Všetky metódy v tejto skupine majú jednu spoločnú vlastnosť a to, že neprijímajú žiadne parametre. Dôvodom k tomu je fakt, že generátor nemôže vedieť, aké hodnoty parametrov by mal pri volaní týchto metód nastaviť. Podobne je to aj s návratovým typom týchto metód, ktorý síce môže byť nastavený ľubovoľne, ale vrátená hodnota sa pri analýze nevyužíva. Z týchto dôvodov odporúčame definovať metódy implementujúce sémantické akcie ako bezparametrické s návratovým typom `void`, aby sa predišlo prípadným nezrovnalostiam. Taktiež odporúčame definovať všetky metódy implementujúce sémantické akcie ako privátne.

Druhú podskupinu tvoria metódy určené k rozšíreniu (verejného alebo privátneho) rozhrania samotného analyzátora. Tieto metódy môžu prijímať ľubovoľné množstvo parametrov a mať ľubovoľný návratový typ, pretože o ich volanie sa stará samotný užívateľ (či už v implementácii samotných sémantických akcií alebo pri práci s objektom triedy *Analyzer*).

Metódy implementujúce sémantické akcie majú spravidla prefix `sa_`.

Poznámka 6.10:
Pri jednotlivých skupinách dynamicky generovaných metód sme vždy spomenuli, že názvy týchto metód sú prefixované špecifickým reťazcom. Dôvodom k tomuto kroku je nutnosť zabezpečiť, aby mená generovaných metód nekolidovali s menami statických[1] metód analyzátora. Táto konvencia zároveň jednoducho umožňuje

[1]Statických v zmysle "nie dynamicky generovaných".

detekovať generovanú metódu podľa jej názvu a priblížiť jej zamýšľané využitie v rámci analyzátora.

Okrem doteraz spomínaných metód je dôležité ešte spomenúť konštruktor a deštruktor triedy *Analyzer*. Nakoľko môže užívateľ v konfiguračnom súbore deklarovať dodatočné členské premenné triedy *Analyzer*, je nutné mu zároveň poskynúť spôsob ako tieto premenné správne inicializovať a prípadne uvoľňovať. Tieto úkony sú taktiež popísané v konfiguračnom súbore a podľa toho je patrične upravená implementácia konštruktora a deštruktora.

Trieda *Analyzer* vždy obsahuje vymenovaný typ *Terminal* a triedu *Lexer* ako privátne vnorené štruktúry. Tento prístup umožňuje oddeliť špecifické časti implementácie medzi viacerými analyzátormi.

Poznámka 6.11:
V závislosti na nastavení (viď nižšie) môže mať trieda *Analyzer* ešte definovanú vnorenú privátnu triedu *Log* spolu s jej inštanciou a zároveň môže byť upravená implementácia niektorých metód.

6.2.10 Logovacie triedy

V tejto sekcii popíšeme dvojicu tried, ktoré spolu zaisťujú zaznamenávanie priebehu analýzy. Tieto triedy sú *BaseLog* a *Log*.

BaseLog

Trieda *BaseLog* je pomocná trieda, ktorej úlohou je zaznamenávať priebeh analýzy do externého súboru. Tento záznam môže byť neskôr využitý pri riešení prípadných problémov a ladení analyzátora. Trieda je spoločná pre všetky vygenerované analyzátory, a preto je definovaná v statickej časti implementácie.

`void setMask(int)`
Samotné zaznamenávanie jednotlivých logovacích správ je závislé na aktuálnej maske logu. Maska je reprezentovaná celým číslom (`int`), ktorého jednotlivé bity určujú, ktoré logovacie správy majú byť zaznamenané. K nastaveniu masky slúži metóda *setMask*, ktorá nastaví masku podľa predaného parametra.

Vytvoriť masku je možné kombináciou (bitovou operáciou `|`) konštánt predefinovaných pre každý typ logovacej správy. V aktuálnej verzii sú definované nasledovné konštanty súvisiace s logovacími správami:

`LOG_VIEW`

Prítomnosť tejto konštanty v maske zabezpečí zaznamenávanie porovnaní terminálneho identifikátora s terminálnym identifikátorom tokenu vo výhľadovom okne, ktoré sú prevádzané pomocou metódy *compareView* triedy *Analyzer*. Typický záznam má nasledovnú podobu:

```
Comparing view at position: 3, expected: T_PLUS, got: T_MINUS.
```

`LOG_NONTERM`

Prítomnosť tejto konštanty v maske zabezpečí zaznamenávanie počiatku a ukončenia rozvoja neterminálu. Typický záznam má tvar:

```
Expanding nonterminal S.
Closing nonterminal.
```

`LOG_TERMINAL`

Prítomnosť tejto konštanty v maske zabezpečí zaznamenávanie konzumácie tokenu z výhľadového okna pomocou metódy *matchTerminal* triedy *Analyzer*. Typický záznam má tvar:

```
Matching terminal T_PLUS ... success.
```

Časť záznamu `"success."` je vypísaná len v prípade, že bol token úspešne spracovaný. V opačnom prípade je záznam ukončený reťazcom `"failed."`.

`LOG_OUTTERM`

Prítomnosť tejto konštanty v maske zabezpečí zaznamenanie zavolania metódy implementujúcej výstupný terminál. Keďže metóda je zvyčajne veľmi jednoduchá (ide len o vypísanie nejakého reťazca na výstup), bolo by zbytočné, aby sa vytváral záznam pri vstupe aj výstupe z tejto metódy. Záznam môže mať teda napríklad nasledujúci tvar:

```
Outterminal outequal method call.
```

`LOG_ACTION`

Prítomnosť tejto konštanty v maske zabezpečí zaznamenanie zavolania užívateľsky definovanej metódy. V tomto prípade implementácia zvyčajne nie je triviálna, a preto nejde efektívne zistiť výstupný bod metódy, ani jej prípadný úspech či neúspech. Záznam sa preto obmedzuje len na vstup do tejto metódy, avšak užívateľ môže prípadné ďalšie potrebné záznamy doplniť v ním poskytnutej implementácii metódy. Štandardný záznam má teda napríklad nasledujúci tvar:

```
Action a_add call.
```

Poznámka 6.12:
Keďže preddefinované konštanty sú definované pre každý analyzátor rovnako, je ich definícia uvedená v statickej časti parseru.

Poznámka 6.13:
Vďaka definovaniu masky logu ako celého čísla môže užívateľ jednoducho definovať vlastné typy správ pre použitie vo vlastnom kóde (užívateľsky definovaných metódach). Jediné čo k tomu potrebuje, je zvoliť si konštantu, ktorá nekoliduje s prednastavenými konštantami a tú potom vo vlastnom kóde použiť pri volaní metódy *logMessage* na preddefinovanej inštancii triedy *BaseLog*.

`int getMask() const`
Táto metóda slúži na získanie aktuálnej masky logu.

`virtual void logMessage(const std::string&, int, bool, bool)`
Metóda *logMessage* sa stará o zapísanie logovacej správy do súboru. Prvý parameter tejto metódy predstavuje samotnú logovaciu správu, ktorá sa má zapísať.

Druhým parametrom je typ správy - predpokladá sa číselná konštanta, ktorej prítomnosť je kontrolovaná v aktuálnej maske logu. V prípade, že sa daná konštanta v maske logu nenachádza, nie je logovacia správa vôbec zapísaná do súboru.

Tretí parameter definuje, či sa má za zapísanú správu vložiť znak predstavujúci koniec riadku. Správy teda nie sú implicitne na samostatnom riadku, pretože jedna správa môže byť zložená z viacerých častí, ktoré nemusia byť v dobe logovania k dispozícii (viď `LOG_TERMINAL`).

Štvrtý parameter definuje, či má byť pred správu vložený prefix vygenerovaný metódou *prefix*.

Metóda je deklarovaná ako virtuálna, pretože sa predpokladá jej predefinovanie v triede *Log*.

`virtual std::string prefix()`

Metóda *prefix* slúži ku generovaniu reťazca, ktorým sú jednotlivé logovacie správy prefixované. V základnej implementácii má tento prefix poskytovať informáciu o hĺbke zanorenia pri rozvoji neterminálov počas priebehu syntaktickej analýzy. S touto metódou súvisia nasledovné dve konštanty, ktoré môžu byť súčasťou masky logu:

`LOG_INDENT_PREFIX`

Pri nastavení tejto hodnoty v maske bude pred logovací záznam vložený reťazec zložený z medzier o dĺžke odpovedajúcej aktuálnemu zanoreniu, čím je možné vytvoriť akúsi stromovú štruktúru logu pripomínajúcu derivačný strom. Pri hlbokom zanorení však môže byť tento prefix nepraktický (pretože bude zbytočne dlhý), a preto je možné použiť nasledovnú možnosť:

`LOG_NUMERIC_PREFIX`

Pri nastavení tejto hodnoty v maske bude pred logovací záznam vložený reťazec obsahujúci hĺbku zanorenia v číselnej podobe.

Metóda je deklarovaná ako virtuálna, pretože sa predpokladá jej prípadne predefinovanie v triede *Log*. Pri predefinovávaní v triede *Log* je vhodné zavolať najskôr pôvodnú implementáciu (z triedy *BaseLog*).

Poznámka 6.14:
Samozrejme je možné skombinovať obe nastavenia prefixu, pričom odsadenie medzerami bude vždy pred číselnou informáciou. Taktiež je možné, aby boli obe nastavenia prefixu vypnuté, čím sa implicitne zabráni vkladaniu prefixu pred logovacie správy.

`void depthShift(int)`

Ako už bolo spomínané, v rámci triedy *BaseLog* je udržiavaná informácia o aktuálnej hĺbke zanorenia pri rozvoji neterminálov. Táto informácia je reprezentovaná typom `unsigned int` a je možné ju meniť pomocou metódy *depthShift*, ktorá prijíma jeden parameter typu *int*, signalizujúci zmenu hĺbky zanorenia. Hodnota tohto parametra je vždy pripočítaná k aktuálnej hodnote, a teda ak sa hĺbka zvyšuje, je hodnota parametra kladná, zatiaľ čo ak sa hĺbka znižuje, je hodnota parametra záporná.

Pri úprave hodnoty zanorenia sa prevádza elementárna kontrola pretečenia. Pri pripočítavaní hodnoty z parametra metódy je kontrolované, že výsledok nebude menší (väčší) ako minimálna (maximálna) hodnota typu `unsigned int`.

V opačnom prípade je nová hodnota nastavená na minimálnu (maximálnu) hodnotu typu `unsigned int`.

`virtual void reset()`
Metóda *reset* slúži na obnovenie nastavení do východzieho stavu. V základnej implementácii to znamená nastavenie aktuálneho zanorenia na 0. Metóda je deklarovaná ako virtuálna, pretože sa predpokladá jej predefinovanie v triede *Log*.

`BaseLog(const std::string& filename)`
V neposlednom rade treba spomenúť konštruktor triedy *BaseLog*. Tento konštruktor prijíma ako jediný parameter názov súboru, do ktorého budú ukladané logovacie záznamy. Súbor s týmto názvom je otvorený pre zápis už v konštruktore a v prípade, že sa tento súbor nepodarí otvoriť, je vyvolaná výnimka. Tento súbor je vždy otvorený s príznakom `std::ios_base::app`, čo zaručí zapisovanie na koniec súboru a dáta, ktoré predtým súbor obsahoval, zostanú zachované.

Okrem vyššie spomínaného ešte konštruktor nastavuje aktuálnu hĺbku zanorenia na hodnotu 0 a nastavuje masku na kombináciu všetkých konštánt typov správ, čo znamená, že všetky implicitne definované typy správ budú zaznamenávané, ale bez prefixu.

Log

Trieda *Log* je deklarovaná ako privátna vnorená trieda triedy *Analyzer*, ktorá dedí funkcionalitu od triedy *BaseLog*. Tento návrh umožňuje užívateľovi zmeniť implementáciu pre všetky vygenerované analyzátory (zmenou implementácie triedy *BaseLog*), ako aj upraviť spôsob logovania pre špecifický analyzátor (predefinovaním implementácie v triede *Log*).

Trieda *Log* v základnej implementácii definuje len konštruktor, ktorý prijíma názov logovacieho súboru a predáva ho konštruktoru triedy *BaseLog*.

Ako už bolo spomenuté v poznámke 6.11, prítomnosť triedy *Log* je závislá na nastavení. Týmto nastavením sa myslí prítomnosť direktívy preprocesoru (`#define`), ktorá má špeciálny tvar zložený z dvoch častí.

Prvá časť (prefix) je tvorená menným priestororom, v ktorom sa nachádza trieda odpovedajúca syntaktickému analyzátoru a samotným názvom tejto triedy, pričom jednotlivé názvy sú od seba oddelené znakom `'_'`. Ak je teda trieda odpovedajúca syntaktickému analyzátoru napríklad:

Test::Analyzers::Analyzer,

bude prefix direktívy odpovedať reťazcu:

`Test_Analyzers_Analyzer`.

Prefix názvu direktívy teda určuje odpovedajúci analyzátor, ktorého sa nastavenie direktívy týka.

Poznámka 6.15:
Názvy direktív sa zvyknú v zdrojovom kóde definovať veľkými písmenami. V tomto prípade to nie je možné (aj napriek autorovým preferenciám), pretože jazyk *C++* rozlišuje medzi veľkými a malými variantami znakov pri deklarovaní štruktúr, a teda je možné definovať rovnaké názvy menných priestorov a tried, ktoré sa

budú líšiť len veľkosťou znakov. Z tohto dôvodu musia varianty znakov v názve direktívy presne odpovedať variantám znakov odpovedajúcich menných priestorov a tried, pretože inak by mohlo dôjsť ku kolíziám v názvoch týchto direktív.

Druhá časť (sufix) názvu direktívy má len informačný charakter. Tvar sufixu je:

```
_LOG_ENABLED
```

a teda celkovo môže mať direktíva napríklad tvar:

```
Test_Analyzers_Analyzer_LOG_ENABLED
```

Táto direktíva slúži ako hlavný vypínač logovania pre daný analyzátor. V prípade, že direktíva s týmto sufixom nie je nastavená, nebude preložená trieda *Log* a v triede *Analyzer* nebude preložená členská premenná `Log * _log` ani metódy *setLogFile* a *setLogMask*.

Poznámka 6.16:
Pozorný čitateľ si môže klásť otázku, prečo je prítomnosť triedy závislá na nastavení direktívy. Dôvod je veľmi jednoduchý - pri použití vygenerovaného analyzátora sa nepočíta s častým zapínaním a vypínaním logovania, a preto je vhodnejšie dať užívateľovi možnosť úplne túto funkcionalitu vylúčiť. Zároveň to prispieva k lepšej efektivite analyzátora, pretože je možné už pri kompilácii vylúčiť všetky volania logovacích metód a program sa nemusí rozhodovať počas behu na základe testovania nejakých nastavení.

Obrázok 6.7 ilustruje vzťahy medzi triedami *LexicalAnalyzer*, *Lexer*, *Analyzer*, *Terminal*, *BaseLog* a *Log*.

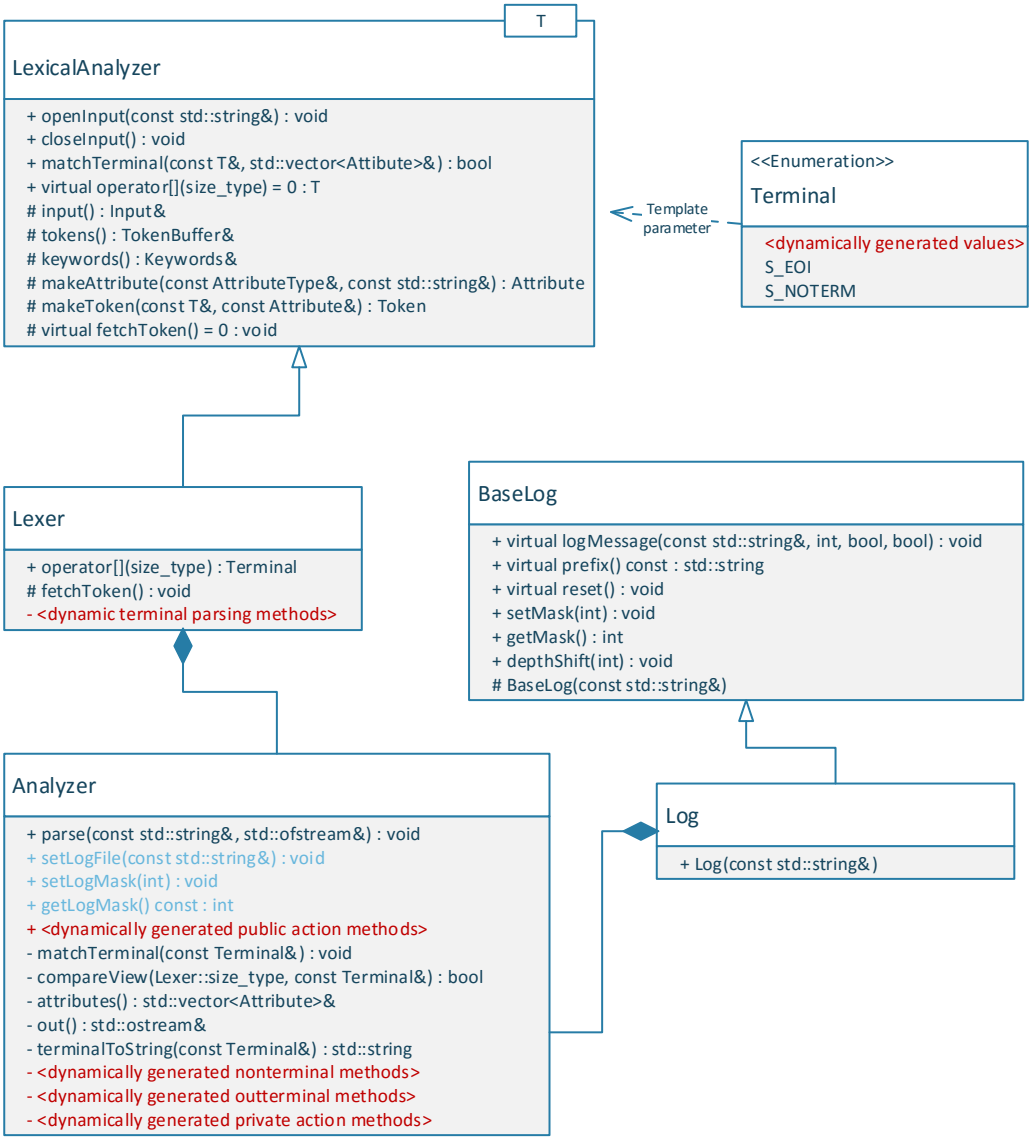
6.2.11 Triedy výnimiek

V rámci analyzátora sú okrem doteraz predstavených tried prítomné ešte ďalšie tri triedy reprezentujúce výnimky. Tieto triedy výnimiek sú odvodené od typu `std::runtime_error` a slúžia na signalizovanie chybových stavov, ktoré môžu nastať počas priebehu analýzy. Každá z týchto tried je používaná v inom kontexte, čím umožňuje jednoduchšie reagovať na rôzne druhy chybových stavov rôznym spôsobom.

internal_error

Prvou výnimkovou triedou je trieda *internal_ error*. Výnimka tohto typu je vyvolaná pri vzniku problému v internej časti analyzátora. Takýmto problémom môže byť napríklad:

- chýbajúce nastavenie,
- chyba pri čítaní vstupu,
- nevalidný stav interných súčastí analyzátora.



Obr. 6.7: UML diagram popisujúci vzťah typu Terminal a tried LexicalAnalyzer, Lexer, Analyzer, BaseLog a Log.

Konkrétnym príkladom kedy je vyvolaná výnimka tohto typu by mohol byť prípad, kedy je vo vygenerovanom analyzátore zapnuté logovanie, ale pred zavolaním metódy *parse* nebola nastavená cesta k súboru, do ktorého majú byť ukladané logovacie správy (teda nebola zavolaná metóda *setLogFile* s validným parametrom). Dá sa teda povedať, že týmto typom sú reprezentované výnimky, ktoré nesúvisia priamo s procesom analýzy.

Trieda reprezentujúca tento typ, je veľmi jednoduchá a odpovedá tomu, čo by užívateľ mohol čakať od štandardného výnimkového typu v rámci jazyka *C++*. To znamená, že poskytuje konštruktor, ktorým je možné nastaviť chybovú správu, ktorá je potom získateľná pomocou metódy *what*.

lexical_error

Ďalšou výnimkovou triedou využívanou v rámci analyzátora je trieda *lexical_ error*. Tento typ výnimky signalizuje problém pri lexikálnej analýze vstupného textu. Týmto problémom môže byť napríklad:

- neschopnosť lexikálneho analyzátora identifikovať token v rámci vstupného textu,
- porušenie nejakého predpokladu, ktorý lexikálny analyzátor očakáva.

Konkrétnym príkladom kedy je vyvolaná táto výnimka by mohla byť situácia, kedy chýba symbol ukončujúci viacriadkový komentár. Táto trieda, podobne ako trieda *internal_ error*, je veľmi jednoduchá a pripomína štandardný výnimkový typ. Poskytuje však dodatočný konštruktor, ktorý umožňuje predať číslo riadku a stĺpca, kde začína problémový vstup. Na základe tejto informácie sa môže vytvoriť generická správa odkazujúca na toto miesto vo vstupnom texte.

syntax_error

Poslednou výnimkovou triedou definovanou v rámci vygenerovaného analyzátora je trieda *syntax_ error*. Výnimka tohto typu signalizuje problém pri syntaktickej analýze. Týmto problémom môže byť napríklad:

- neočakávaný výhľad,
- neodpovedajúci identifikátor predaný metóde *matchTerminal*.

Táto trieda podobne ako predchádajúce odpovedá tomu, čo by užívateľ čakal od štandardného výnimkového typu.

Poznámka 6.17:
Po prečítaní tejto sekcie si môže užívateľ klásť otázku, prečo nie je v rámci analyzátora definovaný typ výnimiek odpovedajúci problému pri sémantickej analýze. Odpoveď na túto otázku je jednoduchá. Sémantická analýza je definovaná výhradne užívateľom poskytnutím kódu jednotlivých sémantických akcií. To znamená, že aj akékoľvek výnimky vyvolané pri sémantickej analýze definuje užívateľ, a preto nie je potrebné definovať ďalší odpovedajúci výnimkový typ.

Poznámka 6.18:
Pri návrhu analyzátora bolo treba rozhodnúť akým mechanizmom budú riešené chybové stavy. Na výber sa ponúkali dve bežne používané techniky: výnimky a chybové kódy. Obe tieto techniky sú aplikovateľné na typ riešeného problému, a preto rozhodnutie bolo výlučne na autorovi.

Zatiaľ čo na počiatku návrhu bolo počítané s použitím chybových kódov, počas samotného vytvárania kódu sa tento prístup rýchlo ukázal ťažkopádny a nevhodný z pohľadu údržby kódu. Pri použití chybových kódov by napríklad z pohľadu návrhu bolo nutné, aby každá metóda nejakým spôsobom vracala svoj výsledný chybový stav (či už vo forme návratovej hodnoty, výstupného parametra alebo pomocou globálnej premennej udržiavajúcej posledný nastavený chybový kód) a to aj v prípade, že je veľmi nepravdepodobné, že použitím danej metódy môže k nejakému chybovému stavu dôjsť.

Ďalšou nevýhodou je, že v prípade použitia chybových kódov je následne tieto chybové kódy neustále nutné kontrolovať, čo podstatne komplikuje a zneprehľadňuje výsledný zdrojový kód, pričom prehľadnosť kódu je jedným z hlavných cieľov návrhu vygenerovaných analyzátorov.

Z týchto dôvodov teda padla voľba na použitie výnimiek.

6.3 Použitie analyzátora

Cieľom tejto časti je poskytnúť jednoduchú funkčnú ukážku použitia vygenerovaného analyzátora. Pôjde o analyzátor odpovedajúci už spomínanej gramatike jednoduchého aritmetického výrazu (viď príklad 3.26), ale príklad je aplikovateľný na akýkoľvek iný analyzátor vygenerovaný programom CppKind.

Program v tomto príklade má za úlohu spustiť analýzu obsahu súboru na základe parametrov predaných cez príkazový riadok a v prípade výskytu nejakého problému vypísať jeho dôvod.

Najskôr uvedieme zdrojový kód programu v tomto príklade a následne tento zdrojový kód podrobne popíšeme.

```
1  #include "_ariexp.h"
2  #include <iostream>
3
4  int main(int argc, char ** argv) {
5    #ifdef Test_Analyzators_AriExp_LOG_ENABLED
6    if (argc != 3) {
7      std::cout << "Invalid number of arguments." << std::endl;
8      std::cout << "USAGE: " << argv[0]
9                << " INPUT_FILE TRACE_FILE" << std::endl;
10
11     return 0;
12   }
13   #else
14   if (argc != 2) {
15     std::cout << "Invalid number of arguments." << std::endl;
16     std::cout << "USAGE: " << argv[0] << " INPUT_FILE" << std::endl;
17
18     return 0;
19   }
20   #endif
21   try {
22     Test::Analyzators::AriExp mp;
23     #ifdef Test_Analyzators_AriExp_LOG_ENABLED
24       mp.setLogFile(argv[2]);
25       mp.setLogMask(LOG_VIEW|LOG_TERMINAL|LOG_OUTTERMINAL|
             LOG_NONTERMINAL|LOG_ACTION);
26     #endif
27     mp.parse(argv[1]);
28   }
29   catch(const std::exception& ex) {
30     std::cout << ex.what() << std::endl;
31   }
32   catch(...) {
33     std::cout << "Unknown exception occured." << std::endl;
34   }
35
36   return 0;
```

```
37 }
```

Listing 6.1: Ukážka použitia vygenerovaného analyzátora.

Aby sme mohli vygenerovaný analyzér použiť, musíme najskôr vložiť hlavičkový súbor, v ktorom je tento analyzér a celá jeho funkcionalita definovaná. V tomto prípade je to súbor `_ariexp.h`. Pre úplnosť ešte vkladáme hlavičkový súbor, ktorý definuje štandardný vstup a výstup (`<iostream>`), ten však k použitiu vygenerovaného analyzéru nie je nutne potrebný.

Ďalej je definovaná metóda *main*, ktorá slúži ako vstupný bod programu používajúci náš vygenerovaný analyzátor. Túto metódu môžeme rozdeliť na dve hlavné časti. Prvá časť (riadky 5 – 20) kontroluje počet parametrov, ktoré boli tejto metóde predané. Samotná kontrola nie je v súvislosti s použitím analyzátora nijak zvlášť zaujímavá, avšak je vhodné všimnúť si kontrolu prítomnosti direktívy `Test_Analyzators_AriExp_LOG_ENABLED`. Ako sme už spomínali, tvar tejto direktívy je odvodený od menného priestoru, v ktorom je deklarovaná trieda implementujúca syntaktický analyzér a samotného názvu tejto triedy.

Táto direktíva môže byť vygenerovaná v hlavičkovom súbore analyzátora. Prítomnosť tejto direktívy značí, že daný analyzátor používa logovanie a očakáva nastavenie cesty k logovaciemu súboru. Táto cesta je v našom príklade predaná programu zadaním parametra na príkazovom riadku, a preto je očakávaný iný počet parametrov než v prípade, že by logovanie nebolo povolené.

V prípade, že počet parametrov predaných programu nie je správny, vypíše program pomocnú správu o predpokladanom použití a ukončí sa. Inak sa pokračuje vykonaním druhej časti programu (riadky 21 – 34), ktorá je tvorená *try-catch* blokom.

V *try* bloku je najskôr vytvorená inštancia vygenerovaného analyzátora. Môžeme si všimnúť, že trieda reprezentujúca tento analyzátor sa nachádza v mennom priestore, ktorý bol špecifikovaný v konfiguračnom súbore.

Následne v závislosti na tom, či je podporované logovanie, je tejto inštancii predaná cesta k súboru, do ktorého bude log ukladaný. Program očakáva túto cestu ako druhý parameter pri spúšťaní na príkazovom riadku. Hodnota tohto parametra je inštancii analyzátora predaná pomocou už spomínanej metódy *setLogFile*. Okrem toho je zároveň nastavená maska, ktorá špecifikuje, ktoré udalosti v rámci analýzy majú byť zaznamenané v logu. Maska je vytvorená kombináciou (pomocou operácie logický OR) preddefinovaných hodnôt a predaná inštancii analyzátora pomocou metódy *setLogMask*. V prípade, že užívateľ nechce definovať vlastnú masku, môže volanie metódy *setLogMask* vynechať a bude použitá prednastavená maska.

Poslednou akciou vykonávanou v *try* bloku je spustenie samotnej analýzy. Program očakáva cestu k súboru, ktorý obsahuje text určený k analýze ako prvý parameter pri spúšťaní na príkazovom riadku. Spustenie analýzy spočíva v zavolaní metódy *parse*, ktorej je predaná táto cesta ako parameter. Metóda *parse* nevracia žiadny výsledok. V prípade, že analýza prebehla bez problémov, je metóda *parse* ukončená a pokračuje sa vo vykonávaní prípadných ďalších inštrukcií. Ak sa počas analýzy vyskytol nejaký problém, je metódou *parse* vyvolaná výnimka.

Nasledujúci *catch* blok odchytáva výnimky typu `std::exception`. Všetky výnimky používané v rámci vygenerovaného analyzátora sú od tohto typu odvodené, a teda budú v tomto bloku zachytené. V prípade zachytenia vypíšeme chybovú

správu na štandardný výstup. Túto chybovú správu získame z odchytenej výnimky použitím metódy *what*.

V rámci tejto ukážky použitia vygenerovaného analyzátora reagujeme na výnimky len vypísaním chybovej hlášky. V prípade, že by sme ale chceli reagovať nejakým sofistikovanejším spôsobom, bolo by vhodné rozlišovať medzi jednotlivými typmi výnimiek, ktoré vygenerovaný analyzátor používa tak, že pridáme ďalšie *catch* bloky odchytávajúce tieto konkrétne typy výnimiek.

Nakoniec je pridaný posledný *catch* blok, ktorého úlohou je odchytiť akúkoľvek výnimku nezachytenú predchádzajúcimi *catch* blokmi a vypísať správu o zachytení neznámej výnimky na štandardný výstup.

Kapitola 7

Záver

Cieľom tejto práce bolo vytvoriť generátor analyzátorov a translátorov založený na prívetivých gramatikách, ktoré zjednodušujú používanie ľavej rekurzie v definícii gramatiky. Generátor mal dokázať pracovať s gramatikami vyžadujúcimi výhľad väčší ako 1, pričom mal čo najviac zachovávať prehľadnosť vygenerovaného zdrojového kódu. Výsledný generátor tieto ciele splňuje.

Okrem toho mal generátor poskytovať podporu importu a exportu nastavení gramatiky aj v iných formátoch než je používaný štandardným konfiguračným súborom. Tento cieľ bol naplnený oddelením mechanizmu importu a exportu nastavení od samotného spracovania vstupnej gramatiky a generovania výsledného analyzátora alebo translátora. Tým vznikla možnosť rozšíriť pôvodný generátor tak, aby mohol využívať v podstate akýkoľvek formát nastavení gramatiky bez toho, aby bolo nutné zasahovať do pôvodného kódu generátora a stačí implementovať samotné načítavanie týchto nastavení.

Okrem splnenia týchto cieľov bol kladený dôraz aj na samotný návrh výsledných analyzátorov a translátorov. Ciele tohto návrhu popísané v úvode tejto práce sa podarilo splniť nasledovne:

1. *Jednoduchosť a prehľadnosť zdrojového kódu*

   Zdrojový kód vygenerovaného analyzátora (translátora) pripomína ručne písaný zdrojový kód (viď sekciu 4.4). Tento kód je pomocou tried rozdelený na ucelené logické celky (viď sekciu 6.1), čo umožňuje rýchlo sa v ňom zorientovať. Pri návrhu boli identifikované statické a dynamické časti analyzátora a tie boli následne rozdelené do samostatných súborov. Pomenovanie jednotlivých prvkov (súborov, tried, metód, premenných, konštánt, ...) bolo volené tak, aby odpovedalo zamýšľanému použitiu a obsahovalo samodokumentačnú hodnotu. Okrem týchto vlastností ďalej k prehľadnosti zdrojového kódu pomáha jednotné formátovanie jednotlivých použitých konštruktov a taktiež samotná dokumentácia väčšiny zdrojového kódu (okrem častí, ktoré pri generovaní nemôžu byť zdokumentované, napr: sémantické akcie) vo formáte vhodnom pre dokumentačný nástroj Doxygen.

2. *Užívateľská príjemnosť*

   Použitie vygenerovaného analyzátora (translátora) je veľmi jednoduché (viď sekciu 6.3). Pre najjednoduchšie prípady je možné vygenerovať zdrojové kódy plne funkčného a spustiteľného programu schopného analyzovať súbor predaný pomocou parametra príkazového riadku. Pri potrebe začlenenia

vygenerovaného analyzátora do väčšieho softwarového celku je možné vygenerovať zdrojové kódy, ktoré sú preložiteľné vo forme samostatnej knižnice alebo je možné zaradiť tieto zdrojové kódy priamo do požadovaného projektu. K samotnému použitiu analyzátora stačí, ak užívateľ vytvorí inštanciu analyzátora a zavolá metódu prevádzajúcu analýzu so správnymi parametrami. Žiadne dodatočné nastavenia nie sú nutné.

3. *Minimálna funkčnosť*

   V základnej forme zabezpečuje vygenerovaný analyzátor len funkcionalitu potrebnú k syntaktickej analýze vstupného textu. Všetka ostatná funkcionalita je dodávaná užívateľom vo forme poskytnutia implementácie sémantických akcií priamo v konfiguračnom súbore (viď sekciu 5.2). Pri implementácii týchto akcií je veľmi jednoduché používať už existujúcu funkcionalitu, nakoľko je možné priamo v konfiguračnom súbore špecifikovať vkladanie (`#include`) externých zdrojových súborov.

4. *Uniformita*

   Vygenerovaný analyzátor sa interne opiera o funkcionalitu šablónových tried, čo umožňuje využívať jednotnú implementáciu pri rôznych nastaveniach (hlavne množiny terminálov). Z toho vyplýva, že vygenerované analyzátory sa v podstate líšia len implementáciou šablónových parametrov, a teda v momente kedy sa užívateľ dokáže orientovať v jednom analyzátore je pre neho jednoduché sa orientovať vo všetkých vygenerovaných analyzátoroch.

   Na základe tejto vlastnosti je implementácia rozdelená na statickú a dynamickú časť reprezentovanú rôznymi súbormi. Toto rozdelenie rovnako umožňuje jednoducho upravovať ako jednotlivé analyzátory, tak celé množiny analyzátorov.

5. *Bezproblémová spolupráca s ďalšími analyzátormi*

   Vďaka využitiu šablónových tried a zapuzdreniu konkrétnych implementačných detailov priamo do triedy reprezentujúcej analyzátor, vystupuje analyzátor ako ohraničená a samostatne funkčná jednotka. To umožňuje, aby vedľa seba fungovalo súčasne niekoľko analyzátorov bez toho, aby dochádzalo k neželaným vedľajším efektom (pri používaní viacerých analyzátorov nad rovnakými dátami súčasne je však funkcionalita obmedzená operačným systémom a všeobecne považovaná za nedefinovanú).

V porovnaní s nástrojom *Bison* generuje nástroj CppKind zrozumiteľnejší a ľahšie overiteľný zdrojový kód, ktorý je možné pomerne jednoducho ďalej modifikovať. Na druhú stranu však nástroj *Bison* umožňuje spracovanie väčšej triedy gramatík.

Oproti nástrojom *Coco/R* a *JavaCC* umožňuje nástroj CppKind použiť v zápise gramatiky priamu ľavú rekurziu, čím sa daná gramatika stáva prirodzenejšou a prehľadnejšou. Zároveň umožňuje nástroj CppKind využívať výhľady väčšie ako 1 bez použitia dodatočných konštruktov ako je tomu pri týchto nástrojoch. Nástroje založené na *LL* gramatikách zároveň vyžadujú, aby pravidlá určujúce nejaký neterminál boli rozlíšiteľné hneď pri vstupe do daného neterminálu

na základe odpovedajúceho výhľadu. Nástroj CppKind založený na prívetivých gramatikách pripúšťa pravidlá s rovnakým neterminálom na ľavej strane a netriviálnym spoločným prefixom ich pravej strany.

7.1 Možnosti ďalšieho rozvoja

Ďalší rozvoj tejto práce by sa mohol uberať niekoľkými smermi. Jedným z nich by mohlo byť napríklad vytvorenie sady "užitočnej funkcionality", o ktorú by sa mohol užívateľ opierať pri používaní vygenerovaných analyzátorov. Táto sada by mohla obsahovať funkcionalitu využiteľnú jednak interne priamo v analyzátore, napríklad obecné funkcie na spracovanie atribútov získaných zo vstupného textu a zároveň funkcionalitu rozširujúcu možnosti použitia analyzátora, napríklad zabezpečujúcu bezproblémovosť použitia analyzátorov vo viacerých vláknach programu.

Druhou možnosťou rozvoja tejto práce by mohlo byť vylepšenie používaného lexikálneho analyzátora. Nakoľko každý vygenerovaný analyzátor má vďaka použitiu šablónových tried definovanú vlastnú konkrétnu implementáciu rozpoznávania tokenov v rámci vstupného textu, nemal by byť problém rozšíriť generátor tak, aby na základe užívateľských nastavení vytváral napríklad implementáciu využívajúcu regulárne výrazy. Zároveň by sa mohla vylepšiť efektivita práce so vstupom. Aktuálne implementovaný lexikálny analyzátor sa snaží držať v pamäti čo najmenšiu časť vstupného textu, to však nemusí byť vhodné pre prácu s veľkými súbormi.

Ďalšou (a podľa autora najzaujímavejšou) možnosťou rozvoja by mohlo byť obohatenie vygenerovaných analyzátorov tak, aby boli schopné reagovať na chybové stavy počas lexikálnej a syntaktickej analýzy a aplikovať obecné metódy ich riešenia alebo metódy špecifikované priamo užívateľom. Pri tomto bode je však nutné podotknúť, že nie je úplne v súlade so zadaním tejto práce, pretože pridaním akejkoľvek logiky spojenej s riešením chybových stavov bude podstatne znížená prehľadnosť a čitateľnosť vygenerovaného zdrojového kódu.

Zoznam použitej literatúry

[1] Alfred V. Aho, Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, Volume I: Parsing. Prentice-Hall, Englewood Cliffs, N.J., 1972. ISBN 0-13-914556-7.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2. vydanie. Addison-Wesley, 2007. ISBN 0-321-49169-6.

[3] Michal Chytil. *Automaty a gramatiky*. 1. vydanie. SNTL, Praha, 1984.

[4] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2. vydanie. Addison-Wesley, 2001 ISBN 0-201-44124-1.

[5] Bořivoj Melichar, Milan Češka, Karel Ježek, Karel Richta. *Konstrukce překladačů*. I. část. Vydavatelství ČVUT, 1999. ISBN 80-01-02028-2.

[6] Bořivoj Melichar, Milan Češka, Karel Ježek, Karel Richta. *Konstrukce překladačů*. II. část. Vydavatelství ČVUT.

[7] Michal Žemlička. *Principles of Kind Parsing*. Dizertačná práca. Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 2006.

[8] Michal Žemlička. *KindCons - Kind Constructor*. http://www.ms.mff.cuni.cz/~zemlicka/KindCons/ 2002.

[9] Michal Žemlička. *KindTrans - Kind Transducer*. http://www.ms.mff.cuni.cz/~zemlicka/KindTran/ 2002.

[10] Pavel Šašek. *Rozširování syntaxe za běhu*. Diplomová práca. Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 2007.

[11] Kateřina Zvánovcová. *Přívětivý translátor v Jave*. Diplomová práca. Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 2008.

[12] Peter Kuziel. *Konštruktor prívetivých analyzátorov pre Javu*. Diplomová práca. Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 2008.

[13] Januš Drózd. *Syntaktická analýza rekurzivním sestupem pro LR(k) gramatiky*. Dizertačná práca, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 1990.

[14] Terrence J. Parr, Russel W. Quong. *LL and LR translators need $k > 1$ lookahead.* ACM SIGPLAN Notices, 31(2):27-34, February 1996.

[15] Albrecht Wöss, Markus Löberbauer, Hanspeter Mössenböck. *LL(1) Conflict Resolution in a Recursive Descent Compiler Generator*. Modular Programming Languages: Joint Modular Languages Conference, JMLC 2003, Klagenfurt, Austria. Pages 192–201. ISBN 978-3-540-45213-3

[16] *CMake* - cross-platform, open-source build system, http://www.cmake.org/

[17] Ken Martin, Bill Hoffman. *Mastering CMake*. 5. vydanie. Kitware, Inc., 2010. ISBN 978-1-930934-22-1

[18] *Yacc* http://dinosaur.compilertools.net/yacc/

[19] *Lex* http://dinosaur.compilertools.net/lex/

[20] *Bison*. https://www.gnu.org/software/bison/

[21] *Coco/R*. http://www.ssw.uni-linz.ac.at/Coco/

[22] *TinyPG*. http://www.codeproject.com/Articles/28294/a-Tiny-Parser-Generator-v

[23] *Grammatica*. https://grammatica.percederberg.net/

[24] *JavaCC*. https://javacc.java.net/

[25] *Flex*. http://flex.sourceforge.net/

Zoznam tabuliek

Zoznam obrázkov

Prílohy

CD s nasledujúcim obsahom:

- *<CD_root_dir>*
  - docs
    - thesis ..... Dokument diplomovej práce
    - documentation ..... Programová dokumentácia aplikácie CppKind
  - examples ..... Ukážky konfiguračných súborov a vstupu
  - sources ..... Zdrojové kódy aplikácie CppKind
  - index-sk.html ..... Informácie k CD v slovenčine
  - index-en.html ..... Informácie k CD v angličtine