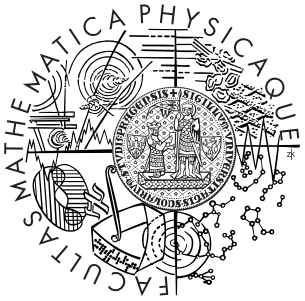
Univerzita Karlova v Prahe
Matematicko-fyzikálna fakulta

# DIPLOMOVÁ PRÁCA




Peter Kuziel

## Konštruktor prívetivých analyzátorov pre Javu

Katedra softwarového inžinierstva

Vedúci diplomovej práce: RNDr. Michal Žemlička, Ph.D.
Študijný program: Informatika

2008

Ďakujem svojmu vedúcemu RNDr. Michalovi Žemličkovi, Ph.D. za cenné rady a pripomienky.

Prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím s požičiavaním práce a jej zverejňovaním.

V Prahe dňa 10. decembra 2008 Peter Kuziel

# Obsah

Názov práce: Konštruktor prívetivých analyzátorov pre Javu
Autor: Peter Kuziel
e-mail: peter.kuziel@seznam.cz
Katedra (ústav): Katedra softwarového inžinierstva
Vedúci diplomovej práce: RNDr. Michal Žemlička, Ph.D.
e-mail vedúceho: zemlicka@ksi.mff.cuni.cz

Abstrakt: Práca predstavuje implementáciu konštruktora analyzátorov, založeného na prívetivých gramatikách. Použitie prívetivých gramatík umožňuje použiť v zápise pravidiel jednoduchú ľavú rekurziu, čo umožňuje pohodlnejšiu prácu s niektorými konštrukciami, ako sú napríklad aritmetické výrazy, ktoré sa pri klasickom prístupe pomocou LL($k$) gramatík formujú menej zrozumiteľne.

Kľúčové slová: prívetivé gramatiky, konštruktor parserov

Title: Kind ParserConstructor for Java
Author: Peter Kuziel
e-mail: peter.kuziel@seznam.cz
Department: Department of software engineering
Supervisor: RNDr. Michal Žemlička, Ph.D.
Supervisor's e-mail address: zemlicka@ksi.mff.cuni.cz

Abstract: The work introduces an implementation of parser constructor, based on kind grammars. Kind grammars allow using simple left recursion in syntactic rules, which makes it easier to work with certain constructions such as arithmetic expressions, that are formulated less comprehensibly when using traditional approach with LL($k$) grammars.

Keywords: kind grammars, parser generator

# Kapitola 1

# Úvod

## 1.1 Zadanie

Vytvorte konštruktor analyzátorov či transducerov z prívetivých gramatík pre Javu. Prihliadnite k ľahkej čitateľnosti a modifikovateľnosti vytvoreného kódu. Zvážte podporu gramatík a analyzátorov s vyhľadom väčším ako 1.

## 1.2 Motivácia

V práci predkladáme program *JKindCons*, ktorý pracuje tak, že na základe gramatiky vygeneruje parser, ktorý môže následne interpretovať vstupný text a vykonávať semantické akcie, ktoré sú súčasťou gramatiky.
Pod pojmom parser rozumieme program, prípadne časť programu, ktorého úlohou je analýza štruktúry vstupných dát a ich transformácia do dátových štruktúr, vhodných na ďalšie spracovanie. Parsery sa v určitej podobe vyskytujú v podstate vo všetkých programoch. Či už sa jedná o jednoduché spracovanie príkazov z príkazového riadka alebo spracovanie zdrojových kódov komplexného programovacieho jazyka. Proces vytvárania parsera je možné automatizovať použitím konštruktora parserov resp. analyzátorov. Konštruktor analyzátorov je program, ktorý na základe špecifikácie štruktúry dát, doplnenej o programátorom definované obslužné rutiny, takzvané akcie, vygeneruje zdrojový kód parsera. Automaticky vytvorená časť kódu realizuje analytickú funkciu parsera, akcie realizujú transformačnú funkciu parsera. Práca nadväzuje na implementáciu prívetivého transducera *KindTran* [6] v jazyku C.

Predkladaný konštruktor analyzátorov je napísaný v jazyku Java, čo okrem iného umožnuje pohodlný zápis akcií, ktoré majú podobu funkcií v jazyku Java.

## 1.3 Štruktúra práce

Text tejto práce je rozdelený do 10 kapitol. Nasledujúca kapitola obsahuje prehľad konštruktorov parserov a lexikálnych analyzátorov. Kapitola 3 obsahuje teoretické poznatky použité pri implementácii prívetivého analyzátora. Kapitola 4 obsahuje prívetivú analýzu. Časť kapitoly je prevzatá zo [7]. V tejto kapitole sa popisuje základná dátová štruktúra parsera. V piatej kapitole sú popisované gramatiky pre praktické použitie – rozšírenie pôvodnej definície gramatiky o sémantické akcie. Táto kapitola je z väčšej časti prevzatá zo [7]. V šietej kapitole je program *JKindCons* popisovaný z užívateľského hľadiska. V tejto kapitole sú tiež implementačné detaily. Kapitola 7 popisuje program *JKindCons*, ktorý môže fungovať aj ako translátor. Kapitola 8 sa zaoberá sémantickými akciami v podobe java kódu, ktoré sú novým prínosom. V tejto kapitole sú uvedené aj implementačné detaily. Kapitola 9 popisuje príbuzné implementácie programu *JKindCons*. V záverečnej kapitole sú zhrnuté výsledky a uvedené možnosti ďalšieho vývoja.

Súčasťou práce je CD, ktoré obsahuje implementáciu programu *JKindCons*. Práca obsahuje jediný dodatok – obsah CD.

# Kapitola 2

# Konštruktory parserov

## 2.1 Základy

Vstupom konštruktora parserov je špecifikácia štruktúry spracovaných dát. Tá sa skladá z nasledujúcich častí:

**Špecifikácia lexikálnej štruktúry dát** – je v niektorých konštruktoroch realizovaná prostredníctvom regulárnych výrazov, kde každý regulárny výraz definuje množinu reťazcov, takzvaných lexémov, ktoré sa môžu vyskytnúť vo vstupných dátach. Ku každému regulárnemu výrazu môže programátor definovať akciu, ktorá sa vykoná keď dôjde k rozpoznaniu lexému, patriaceho do množiny lexémov daného regulárneho výrazu. Rozpoznanie lexému sa často označuje ako zhoda. Kód vygenerovaný z tejto špecifikácie spolu s akciami sa nazýva *lexikálny analyzátor*. Okrem rozpoznávania lexémov a spúšťania akcií, je dôležitou úlohou lexikálneho analyzátora generovanie objektov reprezentujúcich lexémy, takzvaných *tokenov*, ktoré sú spracovávané syntaktickým analyzátorom. Tokeny predstavujú rozhranie medzi lexikálnym a syntaktickým analyzátorom. V závislosti na použitom konštruktore je kód na generovanie tokenov vygenerovaný automaticky alebo je súčasťou akcií. Základným atribútom tokena je jeho identifikátor.

**Špecifikácia syntaktickej štruktúry dát** – je realizovaná prostredníctvom gramatiky. Názvy terminálov odkazujú na identifikátory tokenov, ktoré sú produktom lexikálneho analyzátora. Ku každému pravidlu gramatiky môže programátor definovať akciu, ktorá sa vykoná, keď dôjde k aplikácii pravidla na spracovávané dáta. Kód vygenerovaný z tejto špecifikácie

spolu s akciami sa nazýva *syntaktický analyzátor*. Syntaktický analýzator teda prijíma vstup v podobe tokenov, snaží sa na neho aplikovať pravidlá gramatiky a pri aplikácii pravidla vyvolá zodpovedajúcu akciu.

Oproti naprogramovanému parseru má automaticky vygenerovaný parser nasledujúce výhody:

**Úspora času -** v špecifikácii nie je riešená implementácia parsera. To urýchľuje prácu programátora a znižuje pravdepodobnosť výskytu chýb, ktoré sa ťažko odhaľujú, pretože ich výskyt závisí na povahe vstupných dát.

**Zrozumiteľnosť -** špecifikácia je prístupná každému, kto ovláda gramatiky, ktoré predstavujú intuitívny prístup k popisu štruktúry. Oproti tomu zorientovať sa v konštrukciách naprogramovaného parsera môže byť náročné.

**Znovupoužiteľnosť -** napríklad pri spracovaní zdrojových kódov nejakého programovacieho jazyka bude pravdepodobne voľne dostupná jeho špecifikácia, ktorú stačí doplniť o akcie. Pri programovaní parsera je nutné túto špecifikáciu naprogramovať.

**Modifikovateľnosť -** pri zmene štruktúry dát stačí prepísať špecifikáciu. Naprogramovaný parser je treba preprogramovať.

**Eliminácia nejednoznačnosti -** v prípade, že špecifikácia obsahuje nejednoznačnú gramatiku, je táto skutočnosť programátorovi oznámená. Nejednoznačnosti v naprogramovanom parsere musí programátor odhaliť sám.

## 2.2 Typy konštruktorov parserov

Konštruktory lexikálnych a syntaktických analyzátorov môžu byť implementované ako:

**Samostatné programy -** výhodou rozdelenia do samostatných programov je možnosť definovať v konštruktore lexikálnych analyzátorov rozhranie, ktorým bude lexikálny analyzátor komunikovať so syntaktickým analyzátorom, a tak možnosť vygenerovať z jedného nástroja lexikálne analyzátory, schopné komunikovať s rôznymi syntaktickými analyzátormi.

**Integrované v jednom programe -** pri integrácii do jediného programu je rozhranie pevne dané. Špecifikácia lexikálneho a syntaktického analyzátora je umiestnená v jednom súbore. Niektoré takto implementované konštruktory parserov používajú pre lexikálnu a syntaktickú analýzu rov-

naký algoritmus a umožňujú tak v lexikálnom analyzátore používať prostriedky syntaktickej analýzy.

### 2.2.1 Konštruktory lexikálnych analyzátorov

Vygenerovaný lexikálny analyzátor sa typicky skladá z nasledujúcich častí:

**Ovládač vstupu -** poskytuje ostatným komponentám prístup k znakovým dátam vstupu.

**Výkonná jednotka -** jej úlohou je rozpoznávať vo vstupných dátach lexémy zodpovedajúce zadaným regulárnym výrazom. Väčšinou býva implementovaná ako konečný automat. Na vytvorenie konečného automatu sa používajú štandartné algoritmy z prostredia teórie jazykov. Kód lexikálneho analýzatora, implementovaného pomocou konečného automatu, nie je možné jednoducho modifikovať. Práve z tohto dôvodu existujú generátory, ktoré na implementáciu výkonnej jednotky nepoužívajú konečný automat, ale generujú programové štruktúry, ktoré sú ľahko čitateľné a výsledný lexikálny analyzátor je tak možné jednoducho modifikovať a prispôsobovať.

**Akcie -** kód, ktorý sa vykoná, keď dôjde k rozpoznaniu lexému. Programátor má v akcii prístup k rozpoznanému lexému. V akcii je typicky umiestnený kód na vygenerovanie tokena. Okrem základného atribútu tokena, identifikátora, je ďalším dôležitým atribútom tokena hodnota. Tá je typicky odvodená z lexému.

### 2.2.2 Konštruktory syntaktických analyzátorov

Vstupom konštruktora syntaktických analyzátorov je špecifikácia, ktorá obsahuje pravidlá gramatiky zapísané v notácii BNF (*Backus-Naur Form*), prípadne EBNF (*Extended BNF*) a k nim priradené akcie. Vygenerovaný syntaktický analyzátor pozostáva z nasledujúcich komponent:

**Správa tokenov -** udržuje tokeny získané od lexikálneho analyzátora.

**Výkonná jednotka -** realizuje deriváciu gramatiky podľa analyzovaných dát. V prípade, že existuje viacero výpočtových ciest, je na rozhodnutie o výbere výpočtovej cesty použitá aktuálna sekvencia tokenov na vstupe, takzvaný *výhľad*. V prípade, že nie je možné rozhodnúť podľa výhľadu, je vstupná gramatika pre daný syntaktický analyzátor nejednoznačná. Podľa

implementácie výkonnej jednotky sa v súčastnosti rozdeľujú konštruktory syntaktických analyzátorov na LL($k$) konštruktory a LALR(1) konštruktory.

**Zásobník -** pomocná dátová štruktúra na uchovanie symbolov gramatiky, prípadne stavu analyzátora.

**Akcie -** konštruktor pracuje s dvoma druhmi symbolov: s terminálmi a neterminálmi. Ku každému symbolu je určený názov a typ. Terminály odpovedajú tokenom z lexikálnej analýzy. Názov terminálu zodpovedá identifikátoru tokena a typ terminálu zodpovedá typu hodnoty tokena. V akcii má programátor prístup k hodnotám symbolov na pravej strane pravidla a môže priradiť hodnotu neterminálu na ľavej strane pravidla.

### LL($k$) analyzátory

LL($k$) analyzátory analyzujú vstup zľava doprava a produkujú najľavejšiu deriváciu. Symbol $k$ označuje dĺžku výhľadu v počte tokenov.

Syntaktická analýza začína vložením začiatočného neterminálu na zásobník. V každej ďalšej fáze prebiehajú nasledujúce kroky. V prípade, že je na vrchole zásobníka terminál, porovná sa s aktuálnym tokenom na vstupe. V prípade zhody je ako aktuálny označený nasledujúci token na vstupe a terminál sa zahodí. Inak je oznámená chyba. V prípade, že je na vrchole zásobníka neterminál, nahradí sa pravou stranou pravidla, ktoré má tento neterminál na ľavej strane. V prípade existencie viacerých pravidiel s týmto neterminálom na ľavej strane sa na základe výhľadu musí určiť jedno pravidlo. V prípade, že nie je možné vybrať žiadne pravidlo, je oznámená chyba. Špeciálnym prípadom je LL(0) analyzátor, ktorý zakazuje použitie viacerých pravidiel s rovnakým neterminálom na ľavej strane. Inak by nebolo možné rozhodnúť, ktoré pravidlo použiť.

### LALR(1) analyzátory

LALR(1) analyzátory patria do rodiny LR($k$) analyzátorov. LR($k$) analyzátory analyzujú vstup zľava doprava a produkujú najľavejšiu deriváciu odzadu. Symbol $k$ označuje dĺžku výhľadu v počte tokenov.

Syntaktický analyzátor používa dve akcie. Akciu *posunutia* a *redukcie*. Pri akcii posunutia je aktuálny token na vstupe vložený na zásobník a vstup sa posunie na nasledujúci token. Pri akcii redukcie sa sekvencia terminálov a neterminálov na vrchole zásobníka nahradí neterminálom na ľavej strane pravidla, ktorého pravá strana odpovedá tejto sekvencii. Celá analýza je

realizovaná vykonávaním týchto dvoch akcií. Na rozhodovanie o použití redukcie prípadne posunutia, je použitý konečný automat. Problém nastáva, keď v jednom stave analyzátora je možné súčasne vykonať akciu posunutia a redukcie, prípadne akciu redukcie podľa viacerých pravidiel. V takejto situácii sa musí použiť na rozhodnutie výhľad. Neschopnosť rozhodnúť sa medzi akciou posunutia a redukcie, sa označuje ako *s/r konflikt*. Neschopnosť vybrať pravidlo, podľa ktorého sa akcia vykoná, sa označuje ako *r/r konflikt*. V LR(0) analyzátoroch môže byť v jednom stave analyzátora prípustná iba akcia posunutia alebo redukcie podľa jediného pravidla.

**Porovnanie analyzátorov – všeobecnosť**
LL($k$) gramatiky neumožňujú niektoré konštrukcie zápisu, ktoré sú pomerne bežné pre BNF notáciu. Jedná sa napr. o **ľavú rekurziu**. Je možné odstránenie ľavej rekurzie so zachovaním prostriedkov BNF notácie. Takáto transformácia však väčšinou vedie k neprehľadnej gramatike.

**Porovnanie analyzátorov – jednoduchosť**
Kód analyzátora vygenerovaný z LL($k$) konštruktora je prehľadný a umožňuje jednoduché vykonávanie modifikácií. Vygenerovaný analyzátor sa skladá zo sady metód, kde každá metóda reprezentuje neterminál gramatiky. Vloženie pravej strany pravidla na zásobník je simulované zavolaním metódy, ktorá reprezentuje neterminál na ľavej strane pravidla. Samotný kód rozhoduje, v prípade existencie viacerých pravidiel s rovnakým neterminálom na ľavej strane, o výbere konkretneho pravidla a stará sa o spracovanie pravej strany vybratého pravidla. Teda terminály porovnáva s tokenmi na vstupe a pre neterminály volá odpovedajúce metódy. Veľká časť analyzátora je tak implementovaná použitím konštrukcií programovacieho jazyka.

Oproti tomu LALR(1) analyzátory sú implementované pomocou explicitného zásobníka a automatu, čo je v podstate tabuľka čísel. Takáto implementácia neumožňuje modifikácie a je náročnejšia na pochopenie.

**Porovnanie analyzátorov – akcie**
V LALR(1) analyzátoroch je akcia priradená k pravidlu vykonaná, keď dôjde k použitiu operácie redukcie podľa tohto pravidla. V akcii je programátorovi umožnený prístup k hodnotám terminálov a neterminálov pravidla umiestnených na zásobníku. V akcii môže programátor priradiť hodnotu neterminálu na ľavej strane pravidla. Po vykonaní akcie sa pravá strana pravidla na vrchole zásobníka nahradí neterminálom s priradenou hodnotou na ľavej

strane pravidla.

LALR(1) analyzátory predpokladajú umiestnenie akcií za pravou stranou pravidla. Pre akcie umiestnené v pravidle, konštruktor analyzátorov vykoná jednoduchú transformáciu gramatiky, pri ktorej sú akcie presunuté do automaticky vygenerovaných pravidiel s prázdnou pravou stranou. Umiestnenie akcie do pravidla pri LALR(1) analyzátoroch môže zaviesť do gramatiky nejednoznačnosť.

V LL($k$) analyzátoroch je kód akcie súčasťou tela metódy, ktorá reprezentuje neterminál na ľavej strane pravidla. Hodnoty terminálov a neterminálov sú predané pomocou lokálnych premenných funkcie. Priradenie hodnoty neterminálu na ľavej strane pravidla je realizované priradením návratovej hodnoty metóde.

V LL($k$) analyzátoroch umiestnenie akcií v pravidle nevadí.

## 2.3 Existujúce konštruktory parserov a nástroj JKindCons

**Yacc a Bison**

Nástroje Yacc a Bison [3], [10] sú založené na LALR(1) gramatikách. Pre bezkontextovú gramatiku zapísanú v BNF notácii zkonštruujú parser. Yacc a Bison poskytujú manipuláciu so sémantickým zásobníkom a špecifikáciu sémantických akcií. S každým pravidlom gramatiky môže užívateľ asociovať akcie, ktoré budú vykonané vždy keď dôjde k rozpoznaniu pravidla.

**ANTLR**

Nástroj ANTLR verzia 3.0 [11] je založený na LL(*) gramatikách. Rozdiel medzi LL($k$) a LL(*) gramatikami je uvedený v nasledujúcom príklade. V prípade, že máme 2 pravidlá:

```
input : ID* ';'
input : ID* '.'
```

Tieto pravidlá sú rozlíšiteľné LL(*) parserom, ale nie LL($k$) parserom. Na odlíšenie týchto dvoch alternatív musíme prejsť cez ľubovoľný počet identifikátorov, aby sme videli aký token nasleduje. Nie je tu pevne daný výhľad tokenov dlžky $k$, ktorý bude pracovať.

**CoCo/R**
Nástroj CoCo/R [13] je založený na LL(1) gramatikách. CoCo/R zvláda nielen LL(1) gramatiky. V prípade, že sa vyskytuje LL(1) konflikt, tak CoCo/R rieši konflikty tzv. resolverom, ktorý urobí rozhodnutie o parsovaní na základe viacnásobného výhľadu, alebo sémantickej informácii. Resolver konfliktu je boolean výraz, ktorý je vložený do gramatiky na začiatok prvých dvoch konfliktných alternatív a rozhoduje sa na základe viacnásobného výhľadu či sa táto alternatíva zhoduje s aktuálnym vstupom. V prípade, že resolver odpovie áno, je táto alternatíva zvolená, inak ďalšia alternatíva bude kontrolovaná.

CoCo/R spracúva resolvery konfliktov podobne ako sémantické akcie a jednoducho ich skopíruje do skonštruovaného parsera na tú istú pozíciu na akej sa objavujú v gramatike.

**JavaCC**
Nástroj JavaCC [14] konštruuje parsery, ktoré sú blízke ručne napísaným. JavaCC poskytuje rôzne stratégie na vyriešenie LL(1) konfliktov. JavaCC ponúka techniky na vyriešenie lokálnych konfliktov. Lokálny resolver konfliktov môže byť umiestnený pred alternatívy rovnakým spôsobom ako v CoCo/R. Nasledujúca syntax je dostupná:

- (LOOKAHED($k$) $\alpha|\beta$) hovorí, že parser by sa mal pozrieť na $k$ symbolov dopredu pri pokuse rozpoznať $\alpha$.
- (LOOKAHEAD($boolexpr$) $\alpha|\beta$) dovoľuje užívateľovi špecifikovať boolean výraz. V prípade, že sa výraz vyhodnotí na true, tak pravidlo $\alpha$ je vybraté. Táto stratégia je podporovaná aj nástrojom CoCo/R.

JavaCC poskytuje viac možností na vyriešenie LL(1) konfliktov ako CoCo/R.

**JKindCons**
Nástroj *JKindCons* je založený na prívetivých gramatikách. Generovanie zdrojových kódov parsera pre výhľad dlhší ako 1 nie je priamočiare. Vždy keď by bol použitý dlhší výhľad, korešpondujúce miesto v zdrojovom kóde by bolo ťažšie na čítanie – prinajmenšom neexistuje programová konštrukcia na rozdeľovanie viac nasledujúcich hodnôt súčastne (ako je case/switch príkaz pre jednotlivé hodnoty). Napriek tomu nástroj počíta výhľad do dĺžky $k$ a dokáže rozhodnúť, či daná vstupná gramatika je $k$-prívetivá. Trieda prívetivých gramatík je ostro podtriedou LR gramatík a zároveň ostro nad-

triedou LL gramatík. Platí, že $k$-prívetivé gramatiky generujú LL($k$) jazyky. Podrobnosti a formálne dôkazy je možné nájsť v [7].

# Kapitola 3

# Teória

V tejto kapitole uvedieme niektoré základné pojmy z teórie automatov a gramatík, ktoré budeme ďalej v práci používať a tiež zavedieme značenie. Ďalej sa dostaneme k definícii prívetivých gramatík, čo je práve trieda gramatík, s ktorou vytvorený program *JKindCons* pracuje.

## 3.1 Základné pojmy

**Definícia (Abeceda, slovo, prázdne slovo,** prevzaté z [4], str. 15**):**

**Abecedou** $\Sigma$ sa nazýva konečná neprázdna množina symbolov. **Slovo** nad abecedou $\Sigma$ je konečná postupnoť symbolov abecedy $\Sigma$. V prípade, že je táto postupnosť prázdna, hovoríme o **prázdnom slove** a označujeme ho $\varepsilon$.

**Definícia (Zreťazenie, podslovo, prefix, sufix,** prevzaté z [4], str. 16**):**

Označíme $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ symboly abecedy $\Sigma, u, v, w$ budeme označovať slová. Majme slová:

$$u = a_1 a_2 \ldots a_n,$$

$$v = b_1 b_2 \ldots b_m$$

**Zreťazenie** slov $u$ a $v$ označujeme $uv$ alebo $u \cdot v$ a platí:

$$uv = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m$$

**Podslovom** slova $w$ sa nazýva ľubovolná postupnosť po sebe idúcich symbolov slova $w$.

**Prefixom** slova $w$ je také podslovo $u$ slova $w$, že pre nejaké $v$ platí:

$$w = uv$$

**Sufixom** slova $w$ je také podslovo $v$ slova $w$, že pre nejaké $u$ platí:

$$w = uv$$

$u^*$ označuje množinu slov vzniknutých nula alebo viacej zreťazeniami slova $u$.

V prípade, že je $\Sigma$ abeceda, potom $\Sigma^*$ označuje množinu všetkých slov, vzniknutých zreťazením nula alebo viacej symbolov z $\Sigma$.

**Definícia (regulárny výraz,** prevzaté z [4], str. 72**):**
Buď $\Sigma$ abeceda. Potom definujeme **regulárne výrazy** nasledovne:

- $\emptyset$ (prázdna množina), $\varepsilon$ a $a \in \Sigma$ sú regulárne výrazy. Tieto regulárne výrazy sa nazývajú **primitívne**.
- V prípade, že $r_1, r_2$ sú regulárne výrazy, potom $r_1|r_2$, $r_1r_2$, $r_1^*$ a $(r_1)$ sú tiež regulárne výrazy.
- Slovo je regulárnym výrazom práve vtedy, keď je možné ho vytvoriť z primitívnych regulárnych výrazov konečným počtom aplikácií pravidiel z predchádzajúceho bodu.

**Definícia (jazyk daný regulárnym výrazom,** prevzaté z [4], str. 73**):**
Jazyk $L(r)$ **daný regulárnym výrazom** $r$ je definovaný nasledujúcimi pravidlami:

- $\emptyset$ je regulárny výraz označujúci prázdnu množinu.
- $\varepsilon$ je regulárny výraz označujúci $\{\varepsilon\}$.
- Pre každé $a \in \Sigma$, $a$ je regulárny výraz označujúci $\{a\}$.

V prípade, že $r_1$ a $r_2$ sú regulárne výrazy, potom

- $L(r_1|r_2) = L(r_1) \cup L(r_2) = \{v \in \Sigma^* | v \in L(r_1) \vee v \in L(r_2)\}$.

- $L(r_1 r_2) = L(r_1)L(r_2) = \{w = uv \in \Sigma^* | u \in L(r_1) \wedge v \in L(r_2)\}$.
- $L((r_1)) = L(r_1)$.
- $L(r_1^*) = L(r_1)^* = \{w = v_1 v_2 \ldots v_n \in \Sigma^* | v_1, v_2, \ldots, v_n \in L(r_1) \wedge n \geq 0\}$.

**Definícia (Gramatika,** prevzaté z [4], str. 19**):**
Gramatika $G$ je usporiadaná štvorica $G = (N, \Sigma, P, S)$, kde

- $N$ je konečná, neprázdna množina **neterminálov**,
- $\Sigma$ je konečná, neprázdna množina **terminálov**, množiny $N$ a $\Sigma$ sú disjunktné.
- $P$ je konečná neprázdna množina **pravidiel**. Každé pravidlo má tvar

  $$x \to y,$$

  kde $x, y \in (N \cup \Sigma)^*, x \neq \varepsilon$.
- $S$ je **počiatočný neterminál**. Pri zápise gramatiky ako množiny pravidiel používame konvenciu, že počiatočný neterminál je vždy neterminál na ľavej strane prvého pravidla.

**Definícia (Bezkontextová gramatika,** prevzaté z [4], str. 126**):**
Gramatika $G = (N, \Sigma, P, S)$ je **bezkontextová**, v prípade, že všetky pravidlá z $P$ majú tvar

$$A \to x,$$

kde $A \in N, x \in (N \cup \Sigma)^*$

**Definícia (Derivácia,** prevzaté z [4], str. 20**):**
Majme slovo $w = uxv$ a pravidlo $x \to y$. Hovoríme, že pravidlo $x \to y$ je **aplikovateľné** na slovo $w$ a použitím pravidla $x \to y$ na slovo $w$ vznikne slovo $z = uyv$. Píšeme

$$w \Rightarrow z$$

a hovoríme, že $z$ je **(priamou) deriváciou** $w$.
V prípade, že pre nejaké $n \geq 0$ platí

$$w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n,$$

hovoríme, že $w_n$ je **deriváciou** $w_1$ a zapisujeme

$$w_1 \Rightarrow^* w_n$$

To znamená, že $w_n$ vzniká použitím nula alebo viacej priamych derivácií na $w_1$.

**Definícia (Jazyk generovaný gramatikou,** prevzaté z [4], str. 20**):**
Pre gramatiku $G = (N, \Sigma, P, S)$, nazveme **jazykom gramatiky** $G$ (alebo jazykom **generovaným gramatikou** $G$) množinu

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

**Definícia (Neužitočný symbol,** prevzaté z [4], str. 153**):**
Majme bezkontextovú gramatiku $G = (N, \Sigma, P, S)$. Hovoríme, že symbol $A \in N$ je **neužitočný**, v prípade, že neexistuje $w \in L(G)$ tak, že

$$S \Rightarrow^* xAy \Rightarrow^* w$$

kde $x, y \in (N \cup \Sigma)^*$.

**Definícia (First,** prevzaté z [1], str. 300**):**
$First_G^k(\alpha)$ pre bezkontextovú gramatiku $G$ definujeme ako množinu slov dĺžky $k$ zložených z terminálov, ktorými začínajú slová, ktoré sa derivujú z $\alpha$.

$$First_G^k(\alpha) = \{x \in \Sigma^* | \alpha \Rightarrow^* x\beta, |x| = k \vee (|x| < k \wedge \beta = \varepsilon)\}$$

**Definícia (Follow,** prevzaté z [1], str. 343**):**
$Follow_G^k(\beta)$ pre slovo $\beta$ a bezkontextovú gramatiku $G$ definujeme ako množinu slov dĺžky $k$ zložených z terminálov, ktoré sa môžu objaviť tesne za $\beta$.

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

**Definícia (*A*-Pravidlo,** prevzaté z [7], str. 32**):**
Pravidlo bezkontextovej gramatiky, ktoré má na ľavej strane neterminál $A$, nazveme ***A*-pravidlo**.

**Definícia (Zásobníkový automat,** prevzaté z [4], str. 177**):**
Zásobníkový automat je usporiadaná sedmica

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

kde

- $Q$ je konečná množina **stavov**,
- $\Sigma$ je konečná množina symbolov nazývaná **vstupná abeceda**,
- $\Gamma$ je konečná množina symbolov nazývaná **zásobníková abeceda**,
- $\delta$ je zobrazenie $Q \times (\Sigma \cup \varepsilon) \times \Gamma$ do konečných podmnožín $Q \times \Gamma^*$ a nazýva sa **prechodová funkcia**,
- $q_0 \in Q$ je **počiatočný stav**,
- $z \in \Gamma$ je **počiatočný zásobníkový symbol**,
- $F \subseteq Q$ je množina **koncových stavov**.

**Definícia (konfigurácia, krok zásobníkového automatu**, prevzaté z [4], str. 179**):**
Nech $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ je zásobníkový automat. **Konfigurácia zásobníkového automatu** je usporiadaná trojica $(q, w, u)$, kde

- $q \in Q$ je stav,
- $w \in \Sigma^*$ je zatiaľ neprečítaná časť vstupu,
- $u \in \Gamma^*$ u je obsah zásobníka, symbol na vrchole zásobníka je prvý symbol $u$.

**Krok** z jednej konfigurácie zásobníkového automatu do inej budeme označovať symbolom $\vdash$.
Platí, že krok

$$(q_1, aw, bx) \vdash (q_2, w, yx),$$

kde $q_1, q_2 \in Q$, $w \in \Sigma^*$, $a \in \Sigma$, $x, y \in \Gamma^*$, $b \in \Gamma$ je možný, práve keď

$$(q_2, y) \in \delta(q_1, a, b).$$

**Definícia (Prijímanie zásobníkovým automatom**, prevzaté z [4], str. 180**):**

Hovoríme, že zásobníkový automat $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ **prijíma** slovo $w$, keď sa z konfigurácie $(q_0, w, z)$ dostaneme po konečnom počte krokov do konfigurácie $(q, \varepsilon, x)$, kde $q \in F$ a $x \in \Gamma$.

## 3.2 Prívetivé gramatiky

V tejto kapitole uvedieme pre nás najdôležitejšiu triedu gramatík – $k$ prívetivé gramatiky – čo sú práve gramatiky, ktoré je schopné spracovať konštruktor analyzátorov *JKindCons* popisovaný v tejto práci.

### 3.2.1 Typy produkčných pravidiel

Každé pravidlo ľubovoľnej bezkontextovej gramatiky môže byť priradené do presne jednej z nasledujúcich piatich skupín formálne definovaných nižšie.

1. pravidlá bez ľavej rekurzie (NLRP),
2. pravidlá s priamou ľavou rekurziou (DLRP),
3. pravidlá s nepriamou ľavou rekurziou (ILRP),
4. pravidlá so skrytou ľavou rekurziou (HLRP), a
5. pravidlá so skytou nepriamou ľavou rekurziou (HILRP).

Táto kategorizácia je pevne daná pre ľubovoľnú bezkontextovú gramatiku.

V prípade, že sú v nejakých gramatikách pravidlá z tretej, štvrtej alebo piatej skupiny, gramatiky obsahujúce tieto typy pravidiel sú zle zrozumiteľné a jazyk definovaný touto gramatikou sa ťažko číta. Text prevzatý z [7] str. 32-33.

**Definícia (Ľavá rekurzia, DLRP, NLRP,** prevzaté z [7], str. 32-33**):**

Pravidlo s priamou ľavou rekurziou je pravidlo bezkontextovej gramatiky $G$, ktoré má tvar $A \to A\alpha$, kde $A$ je neterminál a $\alpha \neq \varepsilon$ je reťazec zložený z terminálov a neterminálov.

Množinu všetkých $A$-pravidiel s priamou ľavou rekurziou z bezkontextovej gramatiky $G$ nazveme $DLRP_G(A)$.

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

Množinu všetkých priamo ľavo rekurzívnych pravidiel $G$ nazveme $DLRP_G$ (*directly left-recursive productions*)

$$DLRP_G = \bigcup_{A \in N_G} DLRP_G(A)$$

Pravidlo bez ľavej rekurzie je také pravidlo $A \to \alpha \in P_G$, pre ktoré platí:

$$\forall \beta \in (N_G \cup T_G)^* : \alpha \not\Rightarrow_G^* A\beta$$

Množinu všetkých $A$-pravidiel bezkontextovej gramatiky $G$ bez ľavej rekurzie nazveme $NLRP_G(A)$.

$$NLRP_G(A) = \{A \to \alpha | \forall \beta \in (N_G \cup T_G)^* : \alpha \not\Rightarrow_G^* A\beta\}$$

Množinu všetkých pravidiel $G$ bez ľavej rekurzie nazveme $NLRP_G$ (*non-left recursive productions*)

$$NLRP_G = \bigcup_{A \in N_G} NLRP_G(A)$$

**Definícia (Pravidlá s nepriamou ľavou rekurziou** prevzaté z [7], str. 32-33**):**

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

$$ILRP_G = \bigcup_{A \in N_G} ILRP_G(A)$$

**Definícia (Pravidlá so skrytou ľavou rekurziou** prevzaté z [7], str. 32-33**):**

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

$$HLRP_G = \bigcup_{A \in N_G} HLRP_G(A)$$

**Definícia (Pravidlá so skrytou nepriamou ľavou rekurziou** prevzaté z [7], str. 32-33**):**

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

$$HILRP_G = \bigcup_{A \in N_G} HILRP_G(A)$$

### 3.2.2 Prívetivé gramatiky

**Definícia (DLRF, NLRF** prevzaté z [7], str. 34**):**

Majme bezkontextovú gramatiku $G$. Množinu terminálových slov dĺžky $k$, ktoré môžu nasledovať po neterminále $A$ po jeho ľavej rekurzii, nazveme $DLRF_G^k(A)$(*direct left-recursive follow*).

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

Množinu terminálových slov, ktoré môžu nasledovať po neterminále $A$ mimo jeho ľavú rekurziu nazveme $NLRF_G^k(A)$ (*non-left-recursive follow*).

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

**Definícia (**$k$**-prívetivá gramatika** prevzaté z [7], str. 35**):**

Bezkontextovú gramatiku $G = (N, \Sigma, P, S)$ nazveme $k$**-prívetivou**, v prípade, že

- $P = DLRP_G \cup NLRP_G$
- $\forall A \in N : DLRF_k(A) \cap NLRF_k(A) = \emptyset$
- Pre všetky $A \in N$ platí:
  Pre každé dve pravidlá, ktoré sú obidva z $NLRP(A)$

$$A \to \alpha\beta, A \to \alpha\gamma \in NLRP,$$

kde $\beta$ a $\gamma$ majú najdlhší spoločný prefix $\varepsilon$,
a pre každé dve pravidlá, ktoré sú obidva z $DLRP(A)$

$$A \to \alpha\beta, A \to \alpha\gamma \in DLRP,$$

kde $\beta$ a $\gamma$ majú najdlhší spoločný prefix $\varepsilon$,
platí:

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

Z prvej podmienky definície vyplýva, že $k$-prívetivá gramatika neobsahuje pravidlá so skrytou ľavou rekurziou a nepriamou ľavou rekurziou.

Druhú a tretiu podmienku môžeme neformálne interpretovať tak, že „stačí sa pozrieť na $k$ symbolov dopredu aby sme vedeli čo práve čítame“. Táto informácia je pre parsovanie jazyka veľmi dôležitá a konštrukciu parsera značne uľahčuje.

V prípade niektorých programovacích jazykov sa môže stať napríklad to, že je treba prečítať celý zdrojový text, aby sme boli schopní zistiť, či určitá časť je deklarácia alebo výkonná časť. Analyzátor takého jazyka potom pochopiteľne musí byť komplikovaný, čo zo sebou nesie rad nevýhod.

Naproti tomu prívetivý analyzátor, ktorý popisujeme v tejto práci, má zakladnú štruktúru veľmi jednoduchú a zrozumiteľnú.

**Definícia (Prívetivá gramatika** prevzaté z [7], str. 35**):**
Bezkontextová gramatika $G$ je **prívetivá**, v prípade, že je $k$-prívetivá pre nejaké $k > 0$.

Prívetivé gramatiky generujú LL($k$) jazyky[1] a dovoľujú použitie priamej ľavej rekurzie, čo je výhodné v prípade, keď máme gramatiku, ktorá je prirodzene ľavo rekurzívna – potom nie je nutné ľavú rekurziu odstraňovať, čím by sa gramatika stala horšie zrozumiteľnou.

**Príklad 3.1 (1-Prívetivá gramatika aritmetických výrazov):**
$G = (N, \Sigma, S, P)$
$N = \{S, E, T, F\}$
$\Sigma = \{id, num, (, ), +, *\}$

[1] Dôkaz a ďalšie podrobnosti viď [7] str. 38-66.

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

Podrobnejšie rozobranie vlastností prívetivých gramatík a ich vzťah k ostatným triedam gramatík a vlastností nimi generovaných jazykov, vrátane dôkazov, je možné nájsť v [7].

# Kapitola 4

# Prívetivá analýza

V sekcii o produkčných stromoch sa zoznámime s produkčnými stromami ako s hlavnou dátovou štruktúrou nášho konštruktora parserov a s tým ako ju na základe prívetivej gramatiky vytvoriť.

Popis konštruktora parseru pomocou produkčných stromov je neformálny a závisí do značnej miery na intuícii. Popis pomocou produkčných stromov je ľahko zrozumiteľný.

## 4.1 Produkčné stromy

Les produkčných stromov je jedna z hlavných dátových štruktúr nášho konštruktora parserov. Získame ju veľmi priamočiarym spôsobom z množiny pravidiel prívetivej gramatiky. Postup si vyskúšame na gramatike aritmetických výrazov. Pravidlá najprv rozdelíme do skupín podľa toho, aký neterminál majú na ľavej strane. V rámci týchto skupín ešte rozdelíme pravidlá s priamou ľavou rekurziou a pravidlá bez ľavej rekurzie. (Podľa definície prívetivej gramatiky bude každé pravidlo spadať do jednej z týchto kategórií). Toto rozdelenie ilustruje tabuľka 4.1.

Pretože z pozície pravidla je už jasné, aký neterminál je na ľavej strane, môžeme ho v zápise vynechať. To isté platí i pre prvý neterminál na pravej strane u pravidiel s priamou ľavou rekurziou. Viď tabuľka 4.2.

Tabuľka 4.1: Rozdelenie pravidiel na ľavo rekurzívne a bez ľavej rekurzie

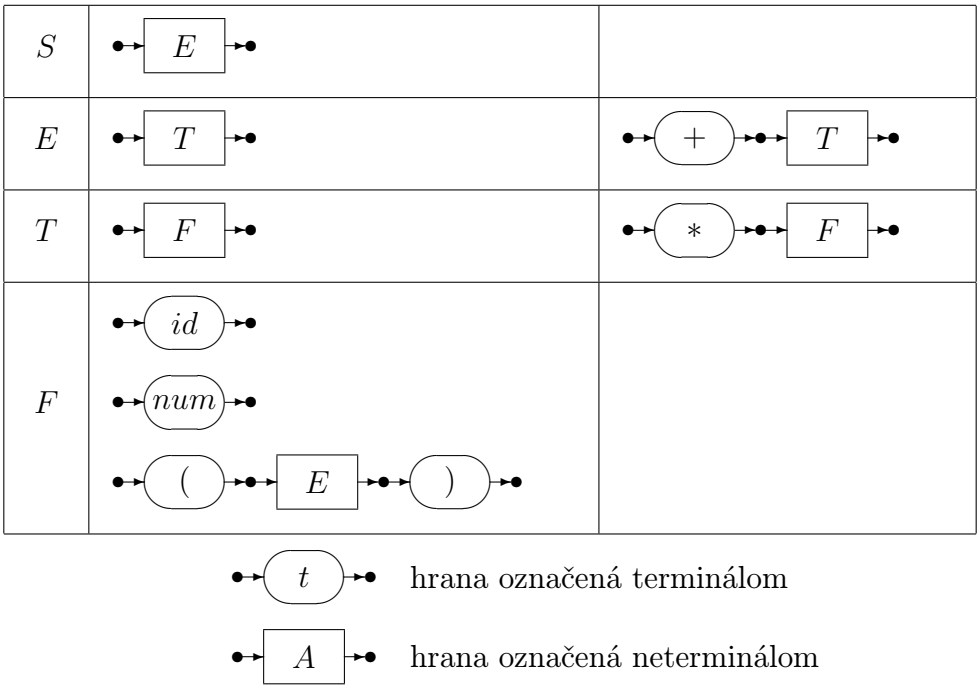
| pravidlá<br>bez ľavej rekurzie | pravidlá<br>s ľavou rekurziou |
|---|---|
| $S \rightarrow E$ | |
| $E \rightarrow T$ | $E \rightarrow E + T$ |
| $T \rightarrow F$ | $T \rightarrow T * F$ |
| $F \rightarrow id$<br>$F \rightarrow num$<br>$F \rightarrow (E)$ | |

Tabuľka 4.2: Rozdelené pravidlá s vynechanou ľavou stranou a ľavo rekurzívnym neterminálom

| ľavá strana<br>– neterminál | pravá strana pravidiel<br>bez ľavej rekurzie | pravá strana pravidiel<br>s ľavou rekurziou |
|---|---|---|
| $S$ | $E$ | |
| $E$ | $T$ | $+T$ |
| $T$ | $F$ | $*F$ |
| $F$ | $id$<br>$num$<br>$(E)$ | |

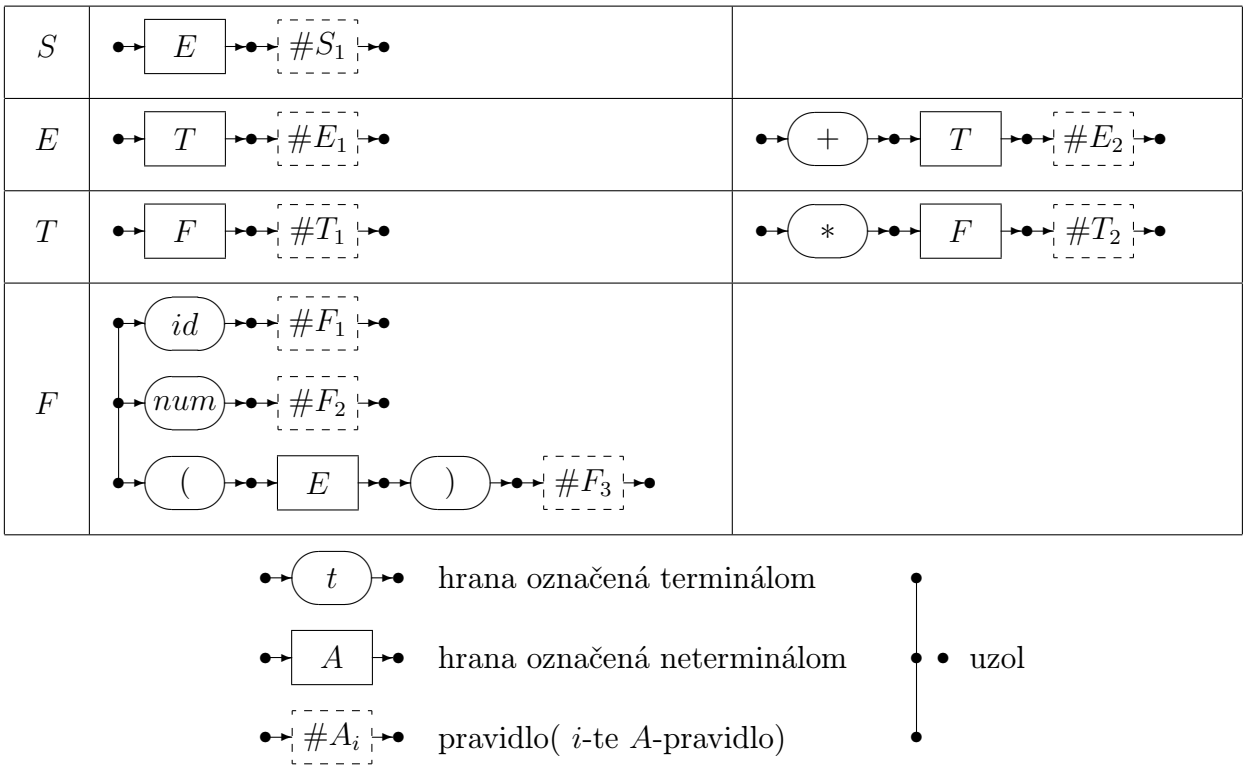
Na takto zapísané pravidlá sa môžeme pozerať ako na cesty v lese produkčných stromov, ako ilustruje tabuľka 4.3. Ku každému neterminálu na ľavej strane bude patriť jeden produkčný strom (v prípade, že pre tento neterminál nie sú žiadne ľavo rekurzívne pravidlá) alebo dva. Hrany v produkčnom strome budú označené symbolmi z pravidiel – hrany na ceste, ktorá zodpovedá danému pravidlu, budú označené postupne práve symbolmi z pravej strany daného pravidla. Výnimkou sú prvé neterminály z pravej strany ľavo rekurzívnych pravidiel, ktorým nezodpovedá žiadna hrana. Inými slovami tieto hrany, ktoré by boli označené ľavo rekurzívnym neterminálom, sú vynechané.

Tabuľka 4.3: Pravidlá ako cesty v lese produkčných stromov




Pre každé pravidlo vytvoríme v lese produkčných stromov naviac ešte jeden list, ktorý bude označovať koniec tohto pravidla. Informácia kde pravidlo končí, je potrebná zvlášť v prípade, že jedno pravidlo je prefixom iného. List, v ktorom končí $n$-té $A$-pravidlo budeme označovať $\#A_n$(viď tabuľku 4.4). Číslovanie sa v našom príklade bude riadiť ich poradím v príklade 3.1.

Tabuľka 4.4: Pravidlá ako cesty v lese produkčných stromov s pridaným listom označujúcim koniec pravidla




## 4.2 Bodkované pravidlá

Keby sme chceli sledovať prácu parseru a držať sa pritom iba popisu pomocou produkčných stromov, bolo by nutné produkčné stromy neustále prekreslovať, čo by viedlo k príliš veľkej priestorovej náročnosti a zlej prehľadnosti. Preto zavedieme ešte iné značenie. Ukážeme, že stavy v lese produkčných stromov sú ekvivalentné triedam obodkovaných pravidiel.

Najprv si zavedieme pojem obodkovaného pravidla[1]. Obodkované pravidlo je pravidlo, ktorého pravá strana obsahuje okrem syntaktických symbolov (terminálov a neterminálov) ešte špeciálny symbol – bodku. Bodka rozdeluje pravú stranu pravidla na dve časti, z nich obidve môžu byť prázdne.

[1]Bodkované pravidlá sa zavádzajú v knihe [2], str 221.

**Príklad 4.1 (Obodkované pravidlo):** $E \to E + \centerdot T$

Obodkované pravidlá sa používajú, neformálne povedané, na rozdelenie pravej strany pravidla na časť už spracovanú a časť ešte nespracovanú. V predchádzajúcom prípade teda časť $E+$ predstavuje už spracovanú časť výrazu (teda bol už prečítaný zo vstupu terminál $+$) zatial čo $T$ je zatiaľ nespracovaná časť.

**Definícia (Prívetivá ekvivalencia obodkovaných pravidiel):**

Buď $G = (N, \Sigma, P, S)$ je bezkontextová gramatika. Hovoríme, že pravidlá $A \to \alpha \centerdot \beta, B \to \gamma \centerdot \delta \in P$ sú **ekvivalentné v prívetivej ekvivalencii**, práve keď $A = B \wedge \alpha = \gamma$ a označujeme

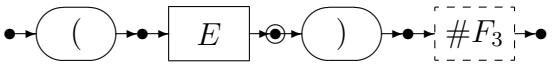
$$A \to \alpha \centerdot \beta =_G B \to \gamma \centerdot \delta$$

Triedy ekvivalencie na pravidlách gramatiky $G$ označujeme $[]_{=G}$ (alebo iba $[]$, keď nehrozí nedorozumenie). Teda

$$A \to \alpha \centerdot \beta =_G B \to \gamma \centerdot \delta \Leftrightarrow A \to \alpha \centerdot \beta \in [B \to \gamma \centerdot \delta]_{=G}$$

Teraz sa vráťme k produkčným stromom. Každý uzol v produkčnom strome leží na nejakej ceste z koreňa do listu. Táto cesta zodpovedá nejakému pravidlu. Vytvoríme obodkované pravidlo tak, že umiestnime bodku medzi symbol, ktorým je označená hrana vedúca do daného uzla, a symbol, ktorým je označená hrana z tohto uzla vychádzajúca. Tak získame pre každý uzol obodkované pravidlo. (Viď obrázok 4.1.)

Obrázok 4.1: Uzol v produkčnom strome a jemu zodpovedajúce obodkované pravidlo




$$E = (E\bullet)$$

Avšak toto zobrazenie nie je vzájomne jednoznačné – môže existovať viac takto vytvorených obodkovaných pravidiel pre jeden uzol. Toto nastane v prípade, že dve pravidlá majú spoločný prefix a bodka je umiestnená v prefixe (presnejšie povedané, keď je bodka pred prvým symbolom, alebo medzi dvoma symbolmi spoločného prefixu, alebo za posledným symbolom prefixu). Teda keď jednému uzlu zodpovedajú dve obodkované pravidlá, potom

majú tvar $A \to \alpha \,.\, \beta$ a $A \to \alpha \,.\, \delta$ a teda sú podľa predchádzajúcej definície ekvivalené, z čoho plynie, že každému uzlu v produkčnom strome zodpovedá práve jedna trieda ekvivalencie obodkovaných pravidiel.

## 4.3 Zásobníkový automat s výhľadom

V tejto časti predstavíme nový typ automatu, ktorý bude základom pre prívetivý analyzátor – zásobníkový automat s výhľadom. Od zásobníkového automatu sa odlišuje tým, že má naviac výhľad, a teda má informáciu nie len o práve čítanom symbole, ale aj o niekoľkých ďalších symboloch, ktoré nasledujú. Týmto symbolom hovoríme výhľad a prechodová funkcia zásobníkového automatu s výhľadom závisí aj na výhľade.

**Definícia (Zásobníkový automat s výhľadom dĺžky k, k prívetivý automat)**
Zásobníkový automat s výhľadom je usporiadaná osmica

$$M = (k, Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

kde

- $k > 0$ je prirodzené číslo udávajúce **dĺžku výhľadu**,
- $Q$ je konečná množina **stavov**,
- $\Sigma$ je konečná množina symbolov nazývaná **vstupná abeceda**,
- $\Gamma$ je konečná množina symbolov nazývaná **zásobníková abeceda**,
- $\delta$ je **prechodová funkcia** $Q \times (\Sigma \cup \varepsilon) \times \Gamma \times \Sigma^k$ do $Q \times \Gamma^*$,
- $q_0 \in Q$ je **počiatočný stav**,
- $z \in \Gamma$ je **počiatočný zásobníkový symbol**,
- $F \subseteq Q$ je množina **koncových stavov**.

**Definícia (konfigurácia, krok zásobníkového automatu s výhľadom):**
Nech $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ je zásobníkový automat. **Konfigurácia** zásobníkového automatu s výhľadom je usporiadaná trojica $(q, w, u)$, kde

- $q \in Q$ je stav,
- $w \in \Sigma^*$ je zatiaľ neprečítaná časť vstupu,
- $u \in \Gamma^*$ je obsah zásobníka, symbol na vrchole zásobníka je prvý symbol $u$.

**Krok** z jednej konfigurácie zásobníkového automatu do inej budeme označovať symbolom $\vdash$.

Platí, že krok

$$(q_1, aa_1a_2 \dots a_k w, bx) \vdash (q_2, a_1a_2 \dots a_k w, yx),$$

kde $q_1, q_2 \in Q$, $w \in \Sigma^*$, $a \in \Sigma$, $x, y \in \Gamma^*$, $b \in \Gamma$ je možný, práve keď

$$(q_2, y) = \delta(q_1, a, b, a_1 \dots a_k).$$

Vždy existuje nanajvyš jedna konfigurácia, do ktorej je možné z vychádzajúcej konfigurácie prejsť.

Nasledujúci algoritmus vytvorí pre danú $k$-prívetivú gramatiku zásobníkový automat s výhľadom dĺžky $k$, ktorý prijíma jazyk generovaný zadanou gramatikou.

**Algoritmus 4.1 (Vytvorenie zásobníkového automatu s výhľadom dĺžky $k$ pre $k$-prívetivú gramatiku**, prevzaté z [7], str. 85**):**
**Vstup:** $k$-prívetivá gramatika $G = (N_G, \Sigma_G, P_G, S_G)$
**Výstup:** zásobníkový automat s výhľadom $M = (k, Q, \Sigma_M, \Gamma, \delta, q_0, z, F)$

1. $Q = \{[A \to \alpha \centerdot \beta]_{=G} | A \to \alpha\beta \in P_G\}$
2. $\Sigma_M = \Sigma_G$
3. $\Gamma = Q \cup \{\circ\}, \circ \notin Q$
4. $q_0 = [S_G \to \centerdot\alpha]_{=G}, S_G \to \centerdot\alpha \in P_G$
5. $Z_0 = \circ$
6. $F = \{[S_G \to \alpha\centerdot]_{=G} | S_G \to \alpha \in P_G\}$

7. $\delta_{scan} =$
$\{([A \to \alpha \centerdot a\beta]_{=G}, a, q, b_1 \dots b_k) \mapsto ([A \to \alpha a \centerdot \beta]_{=G}, q) |$
$A \to \alpha a\beta \in P_G \wedge b_1 \dots b_k \in First^k_G(a\beta \cdot Follow^k_G(A)) \wedge q \in \Gamma\}$

8. $\delta_{descent} =$
$\{([A \to \alpha \centerdot B\beta]_{=G}, \varepsilon, q, b_1 \dots b_k) \mapsto ([B \to \centerdot\gamma]_{=G}, [A \to \alpha B \centerdot \beta]_{=G} \cdot q) |$
$A \to \alpha B\beta, B \to \gamma \in P_G \wedge b_1 \dots b_k \in First^k_G(B\beta \cdot Follow^k_G(A)) \wedge q \in \Gamma\}$

9. $\delta_{recurse} =$
$\{([B \to \alpha\centerdot]_{=G}, \varepsilon, q, a_1 \dots a_k) \mapsto ([B \to B \centerdot \beta]_{=G}, q) |$
$B \to \alpha, B \to B\beta \in P_G \wedge a_1 \dots a_k \in DLRF^k_G(B) \wedge q \in \Gamma\}$

10. $\delta_{finish} =$
$\{([B \to \gamma\centerdot]_{=G}, \varepsilon, [A \to \alpha B \centerdot \beta]_{=G}, a_1 \dots a_k) \mapsto ([A \to \alpha B \centerdot \beta]_{=G}, \varepsilon) |$
$B \to \gamma, A \to \alpha B\beta \in P_G \wedge a_1 \dots a_k \in NLRF^k_G(B)\}$

11. $\delta = \delta_{scan} \cup \delta_{descent} \cup \delta_{recurse} \cup \delta_{finish}$

## 4.4 Použitie produkčných stromov

V produkčných stromoch sme zatiaľ nepracovali s pojmom výhľadu, a preto bude potrebné ich trochu ešte poopraviť, aby sa parser dal jednoducho vygenerovať.

Ako už vieme, uzly v produkčnom strome zodpovedajú triedam ekvivalencie obodkovaných pravidiel. Budeme ich takto teraz označovať.

Symboly na hranách presunieme do uzlov za týmito hranami a hrany označíme zodpovedajúcimi výhľadmi.

- Hrane vedúcej z uzlu $[A \to \alpha \centerdot a\beta]_{=G}$ do uzlu $[A \to \alpha a \centerdot \beta]_{=G}$ priradíme výhľad $First^k_G(a\beta \cdot Follow^k_G(A))$. (Napríklad pre výhľad dĺžky $k = 1$ by bol tento výhľad $a$.)

- Hrane vedúcej z uzlu $[A \to \alpha\centerdot B\beta]_{=G}$ do uzlu $[A \to \alpha B\centerdot\beta]_{=G}$ priradíme výhľad $First^k_G(B\beta \cdot Follow^k_G((A))$.

- Hrane vedúcej z uzlu $[A \to \alpha\centerdot]_{=G}$ do listu označujúceho koniec pravidla $A \to \alpha$ priradíme výhľad $Follow^k_G(A))$.

Tabuľka 4.5: Výhľady zodpovedajúce jednotlivým hranám

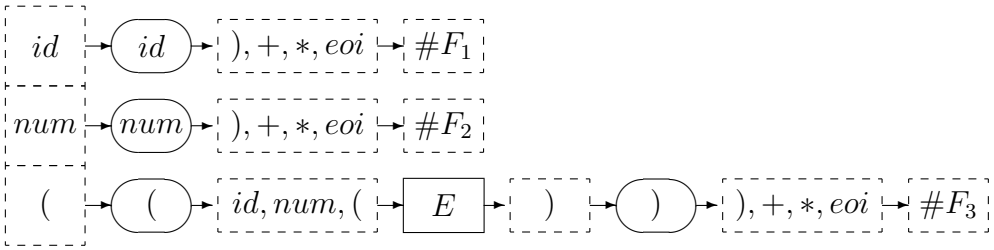
| počiatočný uzol | koncový uzol | výhľad |
|---|---|---|
| $[A \to \alpha \,.\, a\beta]_{=G}$ | $[A \to \alpha a \,.\, \beta]_{=G}$ | $First^k_G(a\beta \cdot Follow^k_G(A))$ |
| $[A \to \alpha \,.\, B\beta]_{=G}$ | $[A \to \alpha B \,.\, \beta]_{=G}$ | $First^k_G(B\beta \cdot Follow^k_G((A))$ |
| $[A \to \alpha .]_{=G}$ | koniec pravidla $A \to \alpha$ | $Follow^k_G(A))$ |

Prehľad označení hrán výhľadmi obsahuje tabuľka 4.5.

Produkčný strom s výhľadmi na hranách zobrazuje obrázok 4.2, špeciálny terminál *eoi* označuje koniec vstupu.

Vygenerovaný parser bude začínať v stave, ktorý zodpovedá koreňu nerekurzívneho stromu počiatočného neterminálu, teda v našom prípade v stave $S \to .E$. Ďalej sa v každom uzle bude rozhodovať podľa výhľadu, kde pokračovať. Práca parseru končí buď zistením, že vstupné slovo nie je korektné slovo generované gramatikou $G$, alebo naopak prijatím zadaného slova. Pokiaľ podľa výhľadu nemáme z aktuálneho stavu ako pokračovať a pritom aktuálny uzol nie je koniec nerekurzívneho pravidla, na ktorého ľavej stane je počiatočný neterminál, alebo nie je prázdny zásobník, je vstupné slovo odmietnuté. Pokiaľ naopak je zásobník prázdny a sme na konci $S$-pravidla (na konci metódy vo vygenerovanom parseri, ktorá reprezentuje neterminál $S$), kde $S$ je počiatočný neterminál, je vstupné slovo prijaté.

Obrázok 4.2: Produkčný strom s výhľadmi na hranách




Ako príklad uvedieme parsovanie výrazu „3+2“. V ľavom stĺpci je zapísaný vstup, pozícia bodky oddeľuje už spracovanú časť od zatiaľ nespracovanej. V druhom stĺpci je vždy aktuálny stav – používame značenie pomocou obodkovaných pravidiel, špeciálny stav zodpovedajúci koncu pravidla (tj. pridanému listu) budeme označovať príslušným pravidlom so znakom „#“ na konci. V treťom stĺpci je zásobník, ktorý je zobrazený tak, že vrchol je hore (teda stavy sa pridávajú na horný koniec a z neho sa tiež odoberajú).

Tabuľka 4.6: Simulácia parsovania výrazu $3 + 2$

| vstup | stav | zásobník | poznámka |
|---|---|---|---|
| $\blacksquare 3+2$ | $S \to \blacksquare E$ | | Začiatok výpočtu, zásobník je prázdny. |
| $\blacksquare 3+2$ | $S \to E\blacksquare$ | | Vykonal sa prechod cez hranu označenú výhľadom $3 \in First_1(E)$. |
| $\blacksquare 3+2$ | $E \to \blacksquare T$ | $S \to E\blacksquare$ | Prechod do nerekurzívneho podstromu neterminálu $E$. |
| $\blacksquare 3+2$ | $E \to T\blacksquare$ | $S \to E\blacksquare$ | Vykonal sa prechod cez hranu označenú výhľadom $3 \in First_1(T)$. |
| $\blacksquare 3+2$ | $T \to \blacksquare F$ | $E \to T\blacksquare$<br>$S \to E\blacksquare$ | Prechod do nerekurzívneho podstromu neterminálu $T$. |
| $\blacksquare 3+2$ | $T \to F\blacksquare$ | $E \to T\blacksquare$<br>$S \to E\blacksquare$ | Vykonal sa prechod cez hranu označenú výhľadom $3 \in First_1(F)$. |
| $\blacksquare 3+2$ | $F \to \blacksquare num$ | $T \to F\blacksquare$<br>$E \to T\blacksquare$<br>$S \to E\blacksquare$ | Prechod do nerekurzívneho podstromu neterminálu $F$. |
| $3\blacksquare+2$ | $F \to num\blacksquare$ | $T \to F\blacksquare$<br>$E \to T\blacksquare$<br>$S \to E\blacksquare$ | Vykonaný prechod cez hranu označenú $num$, prečítanie terminálu 3. |
| $3\blacksquare+2$ | $F \to num\#$ | $T \to F\blacksquare$<br>$E \to T\blacksquare$<br>$S \to E\blacksquare$ | Nie je možné prejsť do nerekurzívneho podstromu |
| $3\blacksquare+2$ | $T \to F\#$ | $E \to T\blacksquare$<br>$S \to E\blacksquare$ | Vykonal sa prechod po hrane označenej výhľadom $+ \in Follow_1(T)$. Nie je možné pokračovať do ľavo rekurzívneho podstromu $T$. |
| $3\blacksquare+2$ | $E \to T\blacksquare$ | $S \to E\blacksquare$ | Stav vyzdvihnutý zo zásobníka. |
| $3\blacksquare+2$ | $E \to T\#$ | $S \to E\blacksquare$ | Vykonal sa prechod po hrane označenej výhľadom $+ \in Follow_1(E)$. Nie je možné pokračovať do ľavo rekurzívneho podstromu $E$. |
| $3\blacksquare+2$ | $E \to E\blacksquare+T$ | $S \to E\blacksquare$ | Vykonal sa prechod do ľavo rekurzívneho podstromu neterminálu $E$. |
| $3+\blacksquare 2$ | $E \to E+\blacksquare T$ | $S \to E\blacksquare$ | Vykonaný prechod cez hranu označenú $+$, prečítanie terminálu $+$ |
| $3+\blacksquare 2$ | $E \to E+T\blacksquare$ | $S \to E\blacksquare$ | Vykonaný prechod cez hranu označenú výhľadom $2 \in First_1(T)$. |

| vstup | stav | zásobník | poznámka |
|---|---|---|---|
| $3 + \bullet 2$ | $T \to \bullet F$ | $E \to E + T\bullet$<br>$S \to E\bullet$ | Vykonaný prechod prechod do nerekurzívneho podstromu neterminálu neterminálu $T$. |
| $3 + \bullet 2$ | $T \to F\bullet$ | $E \to E + T\bullet$<br>$S \to E\bullet$ | Vykonaný prechod cez hranu označenú výhľadom $2 \in First_1(F)$. |
| $3 + \bullet 2$ | $F \to \bullet num$ | $T \to F\bullet$<br>$E \to E + T\bullet$<br>$S \to E\bullet$ | Vykonaný prechod prechod do nerekurzívneho podstromu neterminálu neterminálu $F$. |
| $3 + 2\bullet$ | $F \to num\bullet$ | $T \to F\bullet$<br>$E \to E + T\bullet$<br>$S \to E\bullet$ | Vykonaný prechod cez hranu označenú $num$, prečítanie terminálu 2. |
| $3 + 2\bullet$ | $F \to num\#$ | $T \to F\bullet$<br>$E \to E + T\bullet$<br>$S \to E\bullet$ | Vykonaný prechod po hrane označenej výhľadom $eoi \in Follow_1(F)$. Nie je možné prejsť do rekurzívneho podstromu $F$. |
| $3 + 2\bullet$ | $T \to F\bullet$ | $E \to E + T\bullet$<br>$S \to E\bullet$ | Pravidlo vyzdvihnuté zo zásobníka. |
| $3 + 2\bullet$ | $T \to F\#$ | $E \to E + T\bullet$<br>$S \to E\bullet$ | Vykonaný prechod po hrane označenej výhľadom $eoi \in Follow_1(T)$. Nie je možné prejsť do rekurzívneho podstromu $T$. |
| $3 + 2\bullet$ | $E \to E + T\bullet$ | $S \to E\bullet$ | Pravidlo vyzdvihnuté zo zásobníka. |
| $3 + 2\bullet$ | $E \to E + T\#$ | $S \to E\bullet$ | Vykonaný prechod po hrane označenej výhľadom $eoi \in Follow_1(E)$. Nie je možné prejsť do rekurzívneho podstromu $E$. |
| $3 + 2\bullet$ | $S \to E\bullet$ | | Pravidlo vyzdvihnuté zo zásobníka. |
| $3 + 2\bullet$ | $S \to E\#$ | | Vykonaný prechod po hrane označenej výhľadom $eoi \in Follow_1(S)$. zásobník prázdny, koniec parsovania. |

# Kapitola 5

# Gramatiky pre praktické použitie

V skutočnosti úlohou nie je len rozhodnúť, či daný text patrí do daného jazyka, ale tiež parsovať tento text a použiť parsovanie na preklad. Preto uvedieme gramatiky, ktoré sú vhodné pre tento účel. V konštruktore analyzátorov, ktorý je popisovaný v tejto práci, je povolené akcie umiestňovať skoro kamkoľvek na pravú stranu pravidla – jediné miesto kde akcie byť nemôžu, je pred ľavou rekurziou.

## 5.1 Sémantické akcie

Parsery v ich teoretickej povahe vrátia len výsledok 'áno' alebo 'nie' odpovedajúc na otázku, či daný vstup patrí do daného jazyka.

Pre praktické použitie je lepšie, keď výstup je vo forme syntaktického stromu alebo vo forme reťazcov symbolov reprezentujúcich sémantické akcie, ktoré majú byť vykonané (avšak v správnom poradí).

Takéto parsery vrátia pre vstup v tvare $a_1 a_2 \dots a_n$ výstup v tvare $\sigma_0 a_1 \sigma_1 a_2 \dots a_n \sigma_n$, kde $\sigma_i$ sú reťazce sémantických symbolov. Tieto sémantické akcie môžu byť použité ako vyvolanie činnosti aplikácie, v ktorej parser existuje.

Preto tu budú gramatiky, ktoré majú naviac k terminálnym a neterminálnym symbolom tiež sémantické symboly. Text prevzatý z [7] str. 88.

**Príklad 5.1 (Aritmetické výrazy a sémantické akcie,** prevzaté z [7] str. 88**):**

Keď sú sémantické akcie použité v definícii gramatiky aritmetických výrazov, získame tak nielen syntax popisovaného jazyka, ale aj jeho sémantiku.

Pre jednoduchšie rozpoznanie typu symbolu sú rôzne typy symbolov uvedené v rôznych zátvorkách:

| | |
|---|---|
| () | terminál |
| [] | neterminál |
| {} | sémantická akcia |

Výsledná gramatika so sémantickými akciami je uvedená nižšie:

$[S] \rightarrow [E]\{\text{print}\}$
$[E] \rightarrow [T]$
$[E] \rightarrow [E](\prime+\prime)[T]\{\text{addition}\}$
$[T] \rightarrow [T]$
$[T] \rightarrow [T](\prime*\prime)[F]\{\text{multiplication}\}$
$[F] \rightarrow (id)\{\text{fetch}\}$
$[F] \rightarrow (num)\{\text{fetch}\}$
$[F] \rightarrow (\prime(\prime)[E](\prime)\prime)$

V programe *JKindCons* sa pravidlá gramatiky zadávajú rovnakou syntaxou.

Tabuľka 5.1: Význam sémantických akcií aritmetického výrazu, prevzaté z [7] str. 89

| | |
|---|---|
| {addition} | vyberie dve najvrchnejšie hodnoty zo sémantického zásobníka, sčíta ich a výsledok vloží späť na sémantický zásobník |
| {multiplication} | vyberie dve najvrchnejšie hodnoty zo sémantického zásobníka, znásobí ich a výsledok vloží späť na sémantický zásobník |
| {print} | vyberie najvrchnejšiu hodnotu zo sémantického zásobníka a vypíše ju |
| {fetch} | načíta hodnotu terminálu z lexikálneho analyzátora a vloží ju na sémantický zásobník |

Význam jednotlivých sémantických akcií je popísaný v tabuľke 5.1. Očakáva sa, že tu je sémantický zásobník na ukladanie hodnôt a lexikálny analyzátor je schopný vydať hodnoty terminálov *num* a *id*.

Sémantické akcie v programe *JKindCons* je možné písať na ľubovoľné miesto v pravidle s výnimkou začiatku ľavo rekurzívneho pravidla.

## 5.2 Translačné gramatiky

Podobný význam ako majú sémantické akcie pre parser majú aj výstupné terminálne symboly. Parsery s výstupnou abecedou sa zvyčajne volajú transducery (translátory).

## 5.3 Sémantické translačné gramatiky

**Definícia (Sémantická translačná gramatika,** prevzaté z [7] str. 90**):**
**Sémantická translačná bezkontextová gramatika** je šestica $G = (N, \Sigma, \Phi, \Psi, P, S)$ kde

- $N$ je konečná, neprázdna množina **neterminálov**,
- $\Sigma$ je konečná, neprázdna množina **vstupných terminálnych symbolov** môže byť tiež nazývaná **vstupná abeceda**,
- $\Phi$ je konečná množina **sémantických akcií**,
- $\Psi$ je konečná množina **výstupných terminálnych symbolov** môže byť tiež nazývaná **výstupná abeceda**,
- $P$ je konečná podmnožina $N \times (N \cup \Sigma \cup \Phi \cup \Psi)^*$ nazývaná množina **pravidiel**,
- $S \in N$ je **počiatočný neterminál**, a $N, \Sigma, \Phi$, a $\Psi$ sú navzájom disjunktné.

V prípade, že $\Psi = \emptyset$ potom sa gramatika nazýva **semanticá bezkontextová gramatika**. V prípade, že $\Phi = \emptyset$ potom sa gramatika nazýva **translačná bezkontextová gramatika**.

**Definícia**, prevzaté z [7] str. 90:
Nech $G$ je sémantická translačná bezkontextová gramatika. Bezkontextová gramatika $G'$ sa nazýva **zjednodušenie**, $G$ keď pre všetky symboly $a_i$ z $\Phi$ a $\Psi$ nové neterminály $A_i$ a nové pravidlá $A_i \rightarrow \varepsilon$ sú pridané a všetky

pravidlá obsahujúce $a_i$ sú substituované pravidlami, ktoré obsahujú na tej istej pozícii $A_i$.

Rozdiel medzi výstupnými symbolmi a sémantickými akciami je nasledujúci: prvá abeceda obsahuje symboly generované do výstupu a druhá príkazy na zmenu vnútornej štruktúry (stavu) aplikácie.

**Príklad 5.2 (Transformácia infixu do postfixu s vyhodnotením výrazu**, prevzaté z [7] str. 91**):**

Majme nasledujúcu translačnú gramatiku so sémantickými akciami dovoľujúcu transformovať aritmetický výraz z infixovej notácie do postfixovej notácie s vyhodnotením výrazu.

| | |
|---|---|
| Neterminály | $S, E, T, F$ |
| Vstupné terminály | $num, +, -, *, (, )$ |
| Výstupné terminály | $\underline{+}, \underline{-}, \underline{*}, \underline{=}$ |
| Sémantické akcie | fetch, put, push, pop, add, sub, mul |
| Pravidlá | $S \rightarrow E\underline{=}$ pop put<br>$E \rightarrow E + T\underline{+}$ add<br>$E \rightarrow E - T\underline{-}$ sub<br>$E \rightarrow T$<br>$T \rightarrow T * F\underline{*}$ mul<br>$T \rightarrow F$<br>$F \rightarrow num$ fetch push put<br>$F \rightarrow (E)$ |
| Počiatočný neterminál | $S$ |

Gramatika očakáva, že lexikálny analyzátor ukladá hodnoty $num$ do buffera. Numerické hodnoty sú vyzdvihnuté z buffera pomocou sémantickej akcie "fetch". Je tu tiež sémantický zásobník slúžiaci ako sémantické úložište a špeciálny register – akumulátor.

| Prehľad sémantických akcií | |
|---|---|
| **Názov** | **Popis** |
| fetch | vyzdvihne hodnotu z lexikálneho analyzátora do akumulátora |
| put | vloží hodnotu z akumulátora na výstup |
| push | vloží hodnotu z akumulátora na sémantický zásobník |
| pop | vyberie hodnotu zo sémantického zásobníka do akumulátora |
| add | vyberie dve najvrchnejšie hodnoty zo sémantického zásobníka, sčíta ich a výsledok vloží späť na sémantický zásobník |
| sub | vyberie dve najvrchnejšie hodnoty zo sémantického zásobníka, odčíta ich (najvrchnejšiu a druhú najvrchnejšiu) a výsledok vloží späť na sémantický zásobník |
| mul | vyberie dve najvrchnejšie hodnoty zo sémantického zásobníka, znásobí ich a výsledok vloží späť na sémantický zásobník |

Tieto sémantické akcie (sémantické inštrukcie) sú podporované programom *JKindCons*.

# Kapitola 6

# Java Prívetivý Konštruktor

V tejto časti nájde užívateľ návod na používanie programu *JKindCons*, ktorý je predstavovaný v tejto práci.

Program *JKindCons* sa spúšťa príkazom

```
java -jar jkindcons <languagefile> <outputdirectory>
```

Pre beh programu je nutná verzia prostredia Java Runtime Environment verzia 1.6 alebo vyššia.

Parametre sú nasledujúce:

`<languagefile>` je súbor s lexikálnymi a syntaktickými pravidlami (gramatikou), vrátane sémantických akcií.

`<outputdirectory>` je adresár, do ktorého sa vygeneruje parser.

**Poznámka:** Kódovanie vstupného súboru musí byť UTF-8.

Popis súboru s lexikálnymi aj syntaktickými pravidlami bude uvedený v nasledujúcich častiach.

Program pracuje s 1-prívetivými gramatikami. Generovanie zdrojových kódov parsera pre výhľad dlhší ako 1 nie je priamočiare. Vždy, keď by bol použitý dlhší výhľad, korešpondujúce miesto v zdrojovom kóde by bolo ťažšie na čítanie – pri najmenšom neexistuje programová konštrukcia na rozdeľovanie viac nasledujúcich hodnôt súčasne (ako je case/switch príkaz pre jed-

notlivé hodnoty). Napriek tomu program počíta výhľad do dĺžky $k$ a dokáže rozhodnúť, či daná vstupná gramatika je $k$-prívetivá.

## 6.1 Súbor so syntaktickými a lexikálnymi pravidlami

Celý popis jazyka, z ktorého bude parser skonštruovaný, je uvedený v jednom súbore. Tento súbor je rozdelený do jednotlivých sekcií. Každá sekcia sa venuje špecifickej oblasti popisu jazyka. Prehľad je uvedený v tabuľke 6.1.

Tabuľka 6.1: Prehľad jednotlivých častí v súbore so syntaktickými a lexikálnymi pravidlami

| Meno sekcie | Potrebné | Popis |
|---|---|---|
| `Language` | ano | Nastavenie parametrov jazyka a aplikácie (názvy špeciálnych terminálov, počiatočný neterminál, názov jazyka). Tiež sa tu môže nastaviť v akej podobe sa budú zadavať sémantické akcie – v podobe java kódu, alebo pomocou sémantických inštrukcií. |
| `Terminals` | ano | Zoznam všetkých terminálnych symbolov vrátane špeciálnych. |
| `Nonterminals` | ano | Zoznam všetkých neterminálnych symbolov (vrátane počiatočného neterminálu). |
| `OutTerms` | nie | Zoznam všetkých výstupných terminálnych symbolov. |
| `Actions` | nie | Zoznam všetkých sémantických akcií. |
| `Productions` | ano | Zoznam všetkých pravidiel gramatiky. |
| `Keywords` | nie | Tvary všetkých kľúčových slov. |
| `Symbols` | nie | Tvary všetkých symbolov (terminálnych symbolov obsahujúcich iba nealfanumerické znaky – t.j. "+" alebo ":="). |
| `OutShapes` | nie | Tvary všetkých výstupných terminálnych symbolov, ktoré sú definované v sekcii `OutTerms`. |

| Meno sekcie | Potrebné | Popis |
|---|---|---|
| `Instructions` | nie | Tu sa definujú telá všetkých sémantických akcií pomocou sémantických inštrukcií. Používa sa v prípade, že v sme v sekcii `Language` nastavili sémantické akcie v podobe sémantických inštrukcií. |
| `JavaSemanticActions` | nie | Používa sa v prípade, že v sme v sekcii `Language` nastavili sémantické akcie v podobe java kódu. Táto se má 4 podsekcie, ktoré budú popísané ďalej. |

Premenné v sekcii `Language`:

- `StartingSymbol` – Názov počiatočného neterminálu gramatiky vstupného jazyka. Tento neterminál musí byť definovaný v sekcii `Nonterminals`. Táto premenná je povinná.
- `IdName` – Názov terminálu pre identifikátory. Tento terminál musí byť definovaný v sekcii `Terminals`. V prípade, že názov nie je špecifikovaný, tak sa identifikátory nerozpoznávajú.
- `NumName` – Názov terminálu pre celé čísla. Tento terminál musí byť definovaný v sekcii `Terminals`. V prípade, že názov nie je špecifikovaný, tak sa čísla nerozpoznávajú.
- `CharName` – Názov terminálu pre znaky. Tento terminál musí byť definovaný v sekcii `Terminals`. V prípade, že názov nie je špecifikovaný, tak sa znaky nerozpoznávajú. V prípade použitia oddeľovač znakov `CharDelim` musí byť nastavený.
- `StrName` – Názov terminálu pre reťazce. Tento terminál musí byť definovaný v sekcii `Terminals`. V prípade, že názov nie je špecifikovaný, tak sa reťazce nerozpoznávajú. V prípade použitia oddeľovač reťazcov `StrDelim` musí byť nastavený.
- `EofName` – Názov terminálu pre koniec súboru. Tento terminál musí byť definovaný v sekcii `Terminals`.
- `StrDelim` – Oddeľovač reťazcov. Musí byť zadaný pri používaní reťazcov `StrName`.

- `CharDelim` – Oddeľovač znakov. Musí byť zadaný pri používaní reťazcov `CharName`.
- `AllAheads` – Určuje, či sa majú vytvárať uzly v stromoch pravidiel na všetkých miestach aj v prípade, že tam nie je vetvenie. Hodnota typu `Yes/No`. Implicitné nastavenie `No`.
- `Extended` – Určuje, či sa má používať rozšírená prívetivá gramatika, to znamená oslabenie kontroly konfliktov. V prípade, že je nejaké pravidlo prefixom iného pravidla a je možné pokračovať obidvoma sposôbmi, tak dostane prednosť dlhšie pravidlo. Keď by v tomto prípade nebola oslabená kontrola konfliktov, gramatika by bola odmietnutá už pri počítaní výhľadu.
- `CaseSensitive` – Rozlišuje sa medzi malými a veľkými písmenami pri identifikátoroch a kľúčových slovách vo vstupnom súbore pre vygenerovaný parser? Hodnota typu `Yes/No`. Implicitné nastavenie `No`.
- `SemanticActionsType` – Rozlišuje zápis sémantických akcií v podobe java kódu alebo pomocou sémantických inštrukcií. Povolené hodnoty sú `javaCode/asmInstructions`. Súčasne je možná len jedna varianta.

Odlišnosť formátu vstupného súboru programu *JKindCons* oproti pôvodnej implementácii – programu *KindCons* ([5]) resp. *KindTran* ([6]) je v tom, že program *JKindCons* nepotrebuje poznať rôzne obmedzenia na počet terminálov, neterminálov atď. (obmedzenia na veľkosť štruktúr používaných programom), ktoré je potrebné v programe *KindCons* definovať. Ďalej sa vo vstupnom súbore pre program *JKindCons* môže použiť nová sekcia (`JavaSemanticActions`), ktorá v pôvodnej implementácii nie je.

V sekcii `Terminals` musia byť definované špeciálne terminály, na ktoré sa odkazuje sekcia `Language`. Sú to terminály `IdName, NumName, StrName, CharName, EofName`. Definícia terminálu obsahuje názov terminálu, ktorý je uvedený na jednom riadku. Na veľkosti písmen v názvoch terminálov záleží.

V sekcii `Nonterminals` musí byť definovaný počiatočný symbol, na ktorý odkazuje premenná `StartingSymbol` v sekcii `Language`. Definícia terminálu obsahuje názov terminálu, ktorý je uvedený na jednom riadku. Na veľkosti písmen v názvoch terminálov záleží.

V sekcii `OutTerms` sú výstupné terminály definované na každom riadku jeden. Na veľkosti písmen výstupných terminálov záleží.

V sekcii `Actions` sú názvy sémantických akcií definované na každom riadku jeden. Na veľkosti písmen sematických akcií záleží. Tu sa nachádzajú názvy semantických akcií, ktoré sú definované pomocou asm semantických inštrukcií. V prípade, že sme v sekcii `Language` nastavili premennú `SemanticActionsType` na `asmInstructions` tak môžme použiť túto popisovanú sekciu `Actions`.

V sekcii `Productions` sa nachadzajú pravidlá gramatiky. Každé pravidlo musí byť ukončené bodkočiarkou. Pravidlo sa skladá z ľavej a pravej časti, ktoré sú oddelené reťazcom "`::=`". V pravej časti pravidla musia byť názvy neterminálov uvedené v hranatých zátvorkách `[]`. Názvy terminálov musia byť uvedené v okrúhlych zátvorkách `()`. Názvy semantických akcií musia byť uvedené v zložených zátvorkách {}. Názvy výstupných terminálov musia byť uvedené v špicatých zátvorkách `<>`.

Obrázok 6.1: Tvar definície pravidla

```
neterminál ::= ...[neterminál] ...(terminál) ...{akcia}
               ...<výstupný_terminál> ...;
```

V sekcii `Keywords` sú definované všetky kľučové slová. Na každom riadku je uvedená jedna definícia kľúčového slova. Na veľkosti písmen názvov terminálov záleží. V prípade, že v sekcii `Language` je premenná `CaseSensitive` nastavená na hodnotu `Yes` tak záleží na veľkosti písmen kľúčových slov. V prípade, že má premenná `CaseSensitive` nastavenú hodnotu `No` tak nezáleží na veľkosti písmen kľúčových slov.

Definícia kľučového slova má tvar:

```
terminál = kľúčové slovo
```

Sekcia `Symbols` obsahuje definície symbolov. Symbol môže obsahovať jeden a viac nealfanumerických znakov. Na každom riadku je uvedená jedna definícia symbolu. Zaleží na veľkosti písmen názvov terminálov.

Definícia symbolu má tvar:

```
terminál = symbol
```

V sekcii `OutShapes` sa nachádzajú definície tvarov výstupných terminálov. Na veľkosti písmen výstupného terminálu záleží. Na každom riadku je uvedená jedna definícia.

Definícia tvaru výstupného terminálu má tvar:

```
výstupný terminál = tvar
```

V sekcii `Instructions` sa nachádzajú definície semantických akcií, ktoré sú uvedené v časti `Actions`. Semantické akcie sú definované pomocou sémantických inštrukcií. Pred názvom sémantickej akcie musí byť uvedený znak `#`. Na nasledujúcich riadkoch sa nachádzajú semantické inštrukcie pre túto sémantickú akciu. Príkaz return slúži na ukončenie definície sémantickej akcie. V prípade, že sme v sekcii `Language` nastavili premennú `SemanticActionsType` na `asmInstructions`, tak môžme použiť túto popisovanú sekciu `Instructions`.

Obrázok 6.1: Definícia semantickej akcie

```
#názov_akcie
sémantická inštrukcia
sémantická inštrukcia
⋮
return
```

Sekcia `JavaSemanticActions` má 4 podsekcie:

- `header`
- `nonterminalReturnTypes`
- `declarations`
- `implementations`

V prípade, že nastavíme v sekcii `Language` sémantické akcie v podobe java kódu (nastavením premennej `SemanticActionsType` na hodnotu `javaCode`), tak môžme následne definovať sekciu `JavaSemanticActions` a jej podsekcie. V tomto prípade nemôžeme použiť sekcie `OutTerms`, `OutShapes`, `Actions`, `Instructions`, lebo tieto sekcie sú nahradené podsekciami v sekcii `JavaSemanticActions`.

**Príklad 6.1 (Súbor so syntaktickými a lexikálnymi pravidlami pre jazyk aritmetické výrazy):**

```
[Language]
  Name = DefaultLang
  StartingSymbol = S
  IdName = id
  NumName = num
  EofName = eoi
[Terminals]
  id
  num
  plus
  star
  lpar
  rpar
  eoi
  minus
[Nonterminals]
  S
  E
  T
  F
[Actions]
  getidval
  getnum
  add
  subtract
  multiply
  print
[Productions]
  S::=[E]{print};
  E::=[T];
  E::=[E](plus)[T]{add};
  E::=[E](minus)[T]{subtract};
  T::=[F];
  T::=[T](star)[F]{multiply};
  F::=(id){getidval};
  F::=(num){getnum};
  F::=(lpar)[E](rpar);
[Symbols]
  star = *
  lpar = (
  rpar = )
```

```
plus = +
minus = -
```

**Príklad 6.2 (Vygenerovaný kód parsera):**
Nasledujúce riadky boli vygenerované pre gramatiku v príklade 6.1 programom *JKindCons*.

```
package application.parser;

import java.io.IOException;

import application.actions.Actions;
import application.actions.ActionsImp;
import application.lexer.Lexer;
import application.lexer.LexicalAnalyzer;
import application.lexer.LexicalException;
import application.lexer.TermCode;
import application.lexer.Terminal;

public class Parser implements SyntacticAnalyzer{

  private LexicalAnalyzer lexer;
  private Actions action;

  public Parser(){
    this.lexer = new Lexer();
    this.action = new ActionsImp(lexer);
  }

  /**
   * parsing subroutine for nonterminal S
   */
  private void nont_S()
  throws SyntaxException, LexicalException, IOException {
    Terminal terminal = new Terminal();

    nont_E();
    action.action_print();
  }

  /**
```

```
 * parsing subroutine for nonterminal E
 */
private void nont_E()
throws SyntaxException, LexicalException, IOException {
  Terminal terminal = new Terminal();

  nont_T();
  lexer.lookTerm(terminal);
  while ((terminal.getTermCode() == TermCode.Term_plus)||
  (terminal.getTermCode() == TermCode.Term_minus) ) {
    lexer.lookTerm(terminal);
    switch (terminal.getTermCode()){
      case Term_plus:
        lexer.matchTerm(TermCode.Term_plus);
        nont_T();
        action.action_add();
        break;
      case Term_minus:
        lexer.matchTerm(TermCode.Term_minus);
        nont_T();
        action.action_subtract();
        break;
      default:
        throw new SyntaxException("syntax Error");
    }
    lexer.lookTerm(terminal);
  }
}

/**
 * parsing subroutine for nonterminal T
 */
private void nont_T()
throws SyntaxException, LexicalException, IOException {
  Terminal terminal = new Terminal();

  nont_F();
  lexer.lookTerm(terminal);
  while (terminal.getTermCode() == TermCode.Term_star ) {
    lexer.lookTerm(terminal);
    lexer.matchTerm(TermCode.Term_star);
```

```
      nont_F();
      action.action_multiply();
      lexer.lookTerm(terminal);
    }
  }

  /**
   * parsing subroutine for nonterminal F
   */
  private void nont_F()
  throws SyntaxException, LexicalException, IOException {
    Terminal terminal = new Terminal();

    lexer.lookTerm(terminal);
    switch (terminal.getTermCode()){
      case Term_id:
        lexer.matchTerm(TermCode.Term_id);
        action.action_getidval();
        break;
      case Term_num:
        lexer.matchTerm(TermCode.Term_num);
        action.action_getnum();
        break;
      case Term_lpar:
        lexer.matchTerm(TermCode.Term_lpar);
        nont_E();
        lexer.matchTerm(TermCode.Term_rpar);
        break;
      default:
        throw new SyntaxException("syntax Error");
    }
  }

  public void parse()
  throws SyntaxException, LexicalException, IOException{

    lexer.lexOpen("_input");
    nont_S();
    lexer.lexClose();
  }
}
```

Vygenerovaný kód je blízky ručne písaným parserom. Vygenerovaný kód je ľahký na pochopenie a je ľahko modifikovateľný. Dáva príležitoť zmeniť pôvodnú gramatiku.

## 6.2 Implementačné detaily

Práca programu *JKindCons* môže byť rozdelená do troch fáz:

1. Spracovanie súboru so syntaktickými a lexikálnymi pravidlami.
2. Vytvorenie lesu produkčných stromov pre danú gramatiku.
3. Generovanie tried v jazyku Java, ktoré reprezentujú lexikálny a syntaktický analyzátor a podporu sémantických akcií.

Druhá fáza bola popísaná v kapitole 4, prvá a tretia fáza budú popísané v nasledujúcich častiach.

### 6.2.1 Spracovanie súboru so syntaktickými a lexikálnymi pravidlami

Súbor so syntaktickými a lexikálnymi pravidlami je textový súbor. Tento súbor je rozdelený do sekcií. Každá sekcia začína svojím menom v hranatých zátvorkách. Sekcie sú riadkovo orientované. Aj sekcia `JavaSemanticActions` so svojimi podsekciami. Sekcia `Productions` nie je riadkovo orientovaná. Pravidlá v tejto sekcii sú ukončené bodkočiarkou. Tento súbor sa spracuje po riadkoch až na sekciu `Productions`. Metóda, ktorá spracováva daný súbor, si pamätá aktuálnu sekciu, v ktorej sa nachádza. V prípade, že obsahuje riadok názov sekcie, tak si metóda zmení aktuálnu sekciu.

Metóda, ktorá spracováva riadok, ukladá informácie do triedy `Language`. Pre sekciu `Terminals` sa v triede `Language` vytvoria záznamy o definovaných termináloch. Pre sekciu `Nonterminals` sa v triede `Language` vytvoria záznamy o definovaných neterminálach. Pre sekciu `OutTerms` a `Actions` sa v triede `Language` vyvoria záznamy o definovaných výstupných termináloch a sémantických akciách. Pre sekciu `Productions` sa v triede `Language` vytvorí záznam o pravidle a súčastne sa pravidlo vloží do stromu pravidiel pre daný neterminál. Každý neterminál má dva stromy pravidiel, pre pravidlá s ľavou rekurziou a bez ľavej rekurzie. Pre sekcie `Keywords`, `Symbols`,

`OutShapes` sa dané tvary uložia do triedy `Language`. Pre sekciu `Instructions` sa dané sémantické inštrukcie uložia do triedy `Language`.

Po spracovaní tohto súboru so syntaktickými a lexikálnymi pravidlami bude zavolaná tzv. post-metoda pre sekciu `Productions`, z ktorej sa zavolajú metódy `ViewMakeAll()` a `NodeRebuild()`. Pomocou metódy `ViewMakeAll()` sa vypočítajú výhľady pre všetky neterminály. Využijú sa k tomu stromy pravidiel s ľavou rekurziou a bez ľavej rekurzie. Následne po vypočítaných výhľadoch sa stromy pravidiel doplnia o uzly. To je vykonané pomocou metódy `NodeRebuild()`.

### 6.2.2 Generovanie metódy reprezentujúcej neterminál

Generovanie parsera pre prívetivé gramatiky môže byť rozdelené do oddeliteľných častí – pre každý neterminál sa vygeneruje metóda, ktorá reprezentuje tento neterminál. Je to metóda v jazyku Java.

Štruktúra metódy reprezentujúcej neterminál je komplexná switch konštrukcia, za ktorou môže nasledovať while cyklus, ktorý zvyčajne ako telo má ďalšiu komplexnú switch konštrukciu. Cyklus je generovaný v prípade, že neterminál má aspoň jedno pravidlo s ľavou rekurziou. Text prevzatý z [7] str. 98.

### 6.2.3 Vygenerovaný lexikálny analyzátor

Vo vygenerovanom parseri sa používajú metódy lexikálneho analyzátora. Vygenerovaný parser volá tieto metódy lexikálneho analyzátora `LookTerm()`, `MatchTerm()`. Tieto metódy používajú metódu `FetchTerm()` lexikálneho analyzátora.

Metóda `FetchTerm()` pracuje tak, že prečíta z aktuálnej pozície vo vstupnom súbore pre vygenerovaný parser jeden znak a podľa znaku sa následne rozhoduje.

1. V prípade, že bol prečítaný biely znak tak sa načítaný biely znak preskočí. Opakuje sa to, kým sa nenačíta iný ako biely znak.

2. V prípade, že bol prečítaný koniec súboru a gramatika má definovaný koniec súboru, tak metóda vráti tento terminál. Inak chyba.

3. V prípade, že bol prečítaný znak písmeno a potom prípadne nasledujú ďalšie alfanumerické znaky, tak sa porovná získaný reťazec s kľúčovými

slovami. V prípade zhody metóda vráti príslušný terminál kľúčového slova, inak metóda vráti terminál pre identifikátor a hodnota je uložená v lexikálnom analyzátore. V prípade, že terminál pre idenfikátory nie je definovaný, nastane chyba.

4. V prípade, že bol prečítaný znak číslica, prípadne nasledujú ďalšie číslice a terminál pre celé čísla je definovaný, tak vráti metóda terminál pre celé číslo. Hodnota je uložená v lexikálnom analyzátore.

5. V prípade, že sú povolené reťazce a prečítaný znak je oddeľovač reťazcov, tak sú čítané ďalšie znaky kým sa nenarazí na koniec reťazca. Metóda vráti terminál pre reťazce. V prípade, že sa oddeľovače reťazcov a znakov zhodujú a znaky sú povolené a slovo reťazec má dĺžku jedna, tak metóda vráti terminál pre znaky. Reťazec musí byť ukončený oddeľovačom reťazca. Hodnota je uložená v lexikálnom analyzátore.

6. V prípade, že sú povolené znaky a prečítaný znak je oddeľovač znakov, prečíta sa nasledujúci znak. Ďalej sa očakáva oddeľovač znakov. Vráti metóda terminál pre znaky. Hodnota je uložená v lexikálnom analyzátore.

7. V prípade, že nenastala žiadna z predchádzajúcich možností, tak sa prehľadá komplexná case/switch štruktúra, ktorá reprezentuje symboly. V prípade zhody sa vráti terminál príslušného symbolu.

8. Inak chyba – znak v súbore so vstupným textom pre vygenerovaný parser nezodpovedá žiadnemu terminálu.

Vygenerovaný parser môže volať metódu `FetchVal()` lexikálneho analyzátora, pomocou ktorej vyzdvihne hodnotu spracovaného terminálu.

# Kapitola 7

# Java Prívetivý Translátor

Jednou významnou úlohou softwarového inžinierstva je integrácia niekoľkých samostatných aplikácií. Je zvyčajne urobená odovzdávaním správ. Problémom je, že aplikácie mohli byť nezávisle vyvíjané a pôvodne nie s úmyslom integrácie. Preto používajú rôzne komunikačné jazyky (formáty správ). Vzniká potreba prekladať správy medzi rôznymi komunikačnými jazykmi.

Úloha prekladu medzi rôznymi komunikačnými jazykmi je určená pre samostatné aplikácie tzv. vstupno-výstupné brány (*front-end gates*). V prípade, že komunikačné jazyky sú XML dialekty, tieto vstupno-výstupné brány môžu byť XSLT skripty. V prípade, že komunikačné jazyky nie sú založené na XML, je lepšie použiť transducery (translátory). Text prevzatý z [7] str. 104.

Program *JKindCons* môže plniť úlohu prívetivého translátora. Prívetivé translátory sú založené na sémantických translačných gramatikách.

**Príklad 7.1 (Transformácia aritmetického výrazu z infixovej notácie do postfixovej s vyhodnotením):**

Vstupný súbor pre program *JKindCons*

```
[Language]
  Name = AriConv
  StartingSymbol = S
  IdName = id
  NumName = num
  EofName = eoi
  SemanticActionsType = asmInstructions
[Terminals]
  id
```

```
  num
  plus
  star
  lpar
  rpar
  eoi
  minus
[Nonterminals]
  S
  E
  T
  F
[OutTerms]
  OutPlus
  OutMinus
  OutStar
  OutEqual
[Actions]
  getidval
  getnum
  add
  subtract
  multiply
  print
  space
[Productions]
  S::=[E]<OutEqual>{space}{print};
  E::=[T];
  E::=[E](plus)[T]{add}<OutPlus>{space};
  E::=[E](minus)[T]{subtract}<OutMinus>{space};
  T::=[F];
  T::=[T](star)[F]{multiply}<OutStar>{space};
  F::=(num){getnum};
  F::=(lpar)[E](rpar);
[Symbols]
  star = *
  lpar = (
  rpar = )
  plus = +
  minus = -
[OutShapes]
```

```
  OutPlus = +
  OutMinus = -
  OutStar = *
  OutEqual = =
[Instructions]
  #getnum
  fetch r0
  push r0
  put r0
  spaces 1
  return
  #add
  pop r0
  add s0,r0
  return
  #subtract
  pop r0
  sub s0,r0
  return
  #multiply
  pop r0
  mul s0,r0
  return
  #print
  pop r0
  put r0
  return
  #space
  spaces 1
  return
```

**Príklad 7.2 (Vygenerovaný kód sémantických akcií):**

Nasledujúce riadky boli vygenerované pre gramatiku v príklade 7.2 programom *JKindCons*.

```
package application.actions;

import java.util.Stack;
import application.writer.Writer;
import application.lexer.LexicalAnalyzer;
import application.lexer.ValType;
```

```
import application.lexer.Value;

public class ActionsImp implements Actions{

  private LexicalAnalyzer lexer;
  private Writer writer;

  private Value[] register = new Value[3];
  private Stack<Value> stack = new Stack<Value>();

  public ActionsImp(LexicalAnalyzer lex, Writer writer){
    this.lexer = lex;
    this.writer = writer;

    for(int i=0; i<3; i++){
      register[i] = new Value();
    }
  }

  public void action_getidval(){
    /* no instructions defined; add here your own code */
  }

  public void action_getnum(){
    Value value = new Value();
    Value value1 = new Value();

    /* semantic instruction fetch r0 */
    value = register[0];
    lexer.fetchVal(value);

    /* semantic instruction push r0 */
    value = register[0];
    value1 = new Value();
    value1.setValue(value.getValue());
    value1.setValType(value.getValType());
    stack.push(value1);

    /* semantic instruction put r0 */
    value = register[0];
    if (value.getValType() == ValType.vtChar){
```

```
      writer.getPrintWriter().print("'"+value.getValue()+"'");
    }else if (value.getValType() == ValType.vtStr){
      writer.getPrintWriter().print("\""+value.getValue()+"\"");
    }else {
      writer.getPrintWriter().print(value.getValue());
    }

    /* semantic instruction spaces 1 */
    int tmp = Integer.parseInt("1");
    for (int i=0; i<tmp;i++){
      writer.getPrintWriter().print(" ");
    }

    return;
  }

  public void action_add(){
    Value value = new Value();
    Value value1 = new Value();

    /* semantic instruction pop r0 */
    value1 = stack.pop();
    value = register[0];
    value.setValue(value1.getValue());
    value.setValType(value1.getValType());

    /* semantic instruction add s0,r0 */
    value = stack.get(stack.size()-1);
    Integer tmp = Integer.parseInt(value.getValue())+
    Integer.parseInt(register[0].getValue());
    value.setValue(tmp.toString());

    return;
  }

  public void action_subtract(){
    Value value = new Value();
    Value value1 = new Value();

    /* semantic instruction pop r0 */
    value1 = stack.pop();
```

```
    value = register[0];
    value.setValue(value1.getValue());
    value.setValType(value1.getValType());

    /* semantic instruction sub s0,r0 */
    value = stack.get(stack.size()-1);
    Integer tmp = Integer.parseInt(value.getValue())-
    Integer.parseInt(register[0].getValue());
    value.setValue(tmp.toString());

    return;
  }

  public void action_multiply(){
    Value value = new Value();
    Value value1 = new Value();

    /* semantic instruction pop r0 */
    value1 = stack.pop();
    value = register[0];
    value.setValue(value1.getValue());
    value.setValType(value1.getValType());

    /* semantic instruction mul s0,r0 */
    value = stack.get(stack.size()-1);
    Integer tmp = Integer.parseInt(value.getValue())*
    Integer.parseInt(register[0].getValue());
    value.setValue(tmp.toString());

    return;
  }

  public void action_print(){
    Value value = new Value();
    Value value1 = new Value();

    /* semantic instruction pop r0 */
    value1 = stack.pop();
    value = register[0];
    value.setValue(value1.getValue());
    value.setValType(value1.getValType());
```

```
    /* semantic instruction put r0 */
    value = register[0];
    if (value.getValType() == ValType.vtChar){
      writer.getPrintWriter().print("'"+value.getValue()+"'");
    }else if (value.getValType() == ValType.vtStr){
      writer.getPrintWriter().print("\""+value.getValue()+"\"");
    }else {
      writer.getPrintWriter().print(value.getValue());
    }

    return;
  }

  public void action_space(){
    /* semantic instruction spaces 1 */
    int tmp = Integer.parseInt("1");
    for (int i=0; i<tmp;i++){
      writer.getPrintWriter().print(" ");
    }

    return;
  }

}
```

V prípade, že vygenerovaný parser dostane vstupný súbor uvedený nižšie, tak vydá uvedený výstupný súbor.

**Vstupný súbor:** 4*(3+2)-7*(3-2)

**Výstupný súbor:** 4 3 2 + * 7 3 2 - * - = 13

**Poznámka:** Na začiatku sémantickej akcie sa niekedy vytvorí nový objekt `Value` (referencie `value, value1`) a referencie `value` a `value1` sa následne niekedy prepíšu. Generovanie tohto kódu by sa dalo v budúcej verzii vylepšiť, aby k tomu nedochádzalo.

# Kapitola 8

# Sémantické akcie v podobe java kódu

Sémantické akcie sa môžu zapisovať aj v podobe java kódu. Pravidlo so sémantickou akciou potom môže vyzerať nasledovne:

```
E::=[E](plus)[T]
{System.out.println("tu sa bude vykonávať sčítanie");};
```

Akcie sa môžu vyskytovať v pravidlách kdekoľvek na pravej strane, jedinou výnimkou, kde akcie byť nemôžu, je pred ľavou rekurziou.

V prípade, že chceme v akciách použiť hodnoty terminálov a neterminálov, použijeme špeciálne premenné: pre neterminál na ľavej strane je to premenná s názvom `$$` pre symboly na pravej strane `$`$n$ pre $n+1$ symbol – teda `$0` pre prvý symbol na pravej strane, `$1` pre druhý atď. Napríklad:

```
E::=[E](plus)[T]
{System.out.println("sčítanie"); $$ = $0 + $2; };
```

Akého typu je ktorý neterminál, závisí na tom, akú má návratovú hodnotu. V prípade, že chceme zistiť reťazcovú hodnotu terminálu (t.j. slovo, ktoré sa náčítalo zo vstupu pri spracovaní tohto neterminálu), použijeme funkciu `stringValue()`.

Teda napríklad:

```
F::=(num)
```

```
{System.out.println(" Načítané číslo: " + $0.stringValue());};
```

V prípade, že sa rozhodneme použiť tento spôsob definovania sémantických akcií, tak musíme tiež okrem pravidiel gramatiky vo vstupnom súbore definujúcom jazyk použiť sekciu `JavaSemanticActions`, ktorá má 4 podsekcie.

**podsekcia** `header` môže obsahovať importy rôznych tried. Teda keď sa rozhodneme používať zoznamy, tak tu môžeme napísať

```
import java.util.List;
```

**podsekcia** `nonterminalReturnTypes` v tejto podsekcii musíme definovať návratové hodnoty všetkých neterminálov.

**podsekcia** `declarations` v tejto podsekcii si môžme definovať globálne premenné. Pre inicializáciu je možné použiť špeciálnu funkciu `__init()` viď ďalej.
V deklaráciách nie je povolené použiť žiadne atribúty (`public, protected, private, static, final`).

**podsekcia** `implementations` tu je možné definovať funkcie, ktoré sa budú používať v akciách – nie je teda potrebné zapísať celú akciu priamo v pravidle, ale môžeme si zavolať funkciu definovanú v tejto podsekcii. Špeciálne premenné `$$`, `$`$n$ je však možné používať iba priamo v pravidlách. Špeciálna premenná nemôže odkazovať na položku v pravidle, ktorá je uvedená za sémantickou akciou, v ktorej je táto špeciálna premenná. Okrem funkcií, ktoré sa použijú v rámci akcií, ešte musíme definovať funkciu `__init()` (bez parametrov), ktorá sa volá automaticky pred začiatkom parsovania.

**Príklad 8.1 (Súbor s gramatikou so sémantickými akciami v podobe java kódu):**

```
[Language]
  Name = AriExpJ
  StartingSymbol = S
  IdName = id
  NumName = num
  EofName = eoi
```

```
  SemanticActionsType = JavaCode
[Terminals]
  id
  num
  plus
  star
  lpar
  rpar
  eoi
  minus
[Nonterminals]
  S
  E
  T
  F
[Productions]
  S::=[E]{System.out.print(outEqual + " " + $0);};
  E::=[T]{$$ = $0;};
  E::=[E](plus)[T]{$$ = scitaj( $0, $2);};
  E::=[E](minus)[T]{$$ = odcitaj( $0, $2);};
  T::=[F]{$$ = $0;};
  T::=[T](star)[F]{$$ = vynasob( $0, $2);};
  F::=(num){$$ = Integer.parseInt( $0.stringValue()); };
  F::=(lpar)[E](rpar){$$ = $1;};
[Symbols]
  star = *
  lpar = (
  rpar = )
  plus = +
  minus = -
[JavaSemanticActions]
[[header]]

import java.util.List;
import java.util.ArrayList;

[[nonterminalReturnTypes]]
  void S
  int E
  int T
  int F
```

```
[[declarations]]

String outEqual;

[[implementations]]

/* Funkcia na scitanie dvoch argumentov */
int scitaj(int arg1, int arg2)
{
  int i = arg1+arg2;
  return i;
}

/* Funkcia na odcitanie dvoch argumentov */
int odcitaj(int arg1, int arg2)
{
  int i = arg1-arg2;
  return i;
}

/* Funkcia na vynasobenie dvoch argumentov */
int vynasob(int arg1, int arg2)
{
  int i = arg1*arg2;
  return i;
}

/* Funkcia __init sa automaticky zavola pred zaciatkom parsovania */
public void __init()
{
  System.out.println("-- Vypocet hodnoty vyrazu --");
  outEqual = "=";
}
```

## 8.1 Popis implementácie

V prípade, že použijeme java sémantické akcie, tak sa na základe súboru s gramatikou vygeneruje parser, pre ktorý platí nasledovné:

- obsahuje všetky premenné, ktoré sú deklarované v podsekcii `declarations`;

- pre každý terminál $x$, ktorý sa vyskytuje maximálne $n$-krát v nejakom $A$-pravidle, sa vygenerujú položky $term\_x\_1, \ldots, term\_x\_n$ v metóde, ktorá reprezentuje neterminál $A$;
- pre každý neterminál $X$, ktorý sa vyskytuje maximálne $n$-krát v nejakom $A$-pravidle, sa vygenerujú položky $nont\_X\_1, \ldots, nont\_X\_n$ v metóde, ktorá reprezentuje neterminál $A$;
- vygenerované položky pre terminály a neterminály majú prefix $term\_$ a $nont\_$, aby sa zamedzilo vzniku kolízií medzi terminálmi a neterminálmi;
- vygenerované položky pre terminály (neterminály) majú medzi názvom terminálu (neterminálu) a číslom znak '_', aby sa zamedzilo kolíziám. V prípade, že by tam znak '_' nebol, tak by príslušné terminály (neterminály) mohli kolidovať, napr. názov terminálu $x$ by mohol byť v kolízii s názvom terminálu $x1$;
- obsahuje metódu pre každú funkciu definovanú v podsekcii `implementations`;
- obsahuje telo každej sématickej akcie, ktoré je v parsere vygenerované na správnom mieste a miesto premenných $\$\$, \$0, \ldots, \$n$ používa položky, ktoré zodpovedajú príslušným symbolom z pravidiel;
- Špeciálna premenná nemôže odkazovať na položku v pravidle, ktorá je uvedená za sémantickou akciou, v ktorej je táto špeciálna premenná;
- obsahuje deklarácie balíkov a importy, ktoré sú uvedené v sekcii `header`;
- v prípade, že 2 pravidlá v sémantickej akcii zapisujú do tej istej globálnej premennej, tak pri opätovnom použití pravidla dôjde k prepísaniu premennej.

Nasledujúci príklad ukazuje triedu vygenerovanú z gramatického súboru z príkladu 8.1.

**Príklad 8.2 (Vygenerovaná trieda Parser):**

```
package application.parser;

import java.io.IOException;
```

```
import application.lexer.Lexer;
import application.lexer.LexicalAnalyzer;
import application.lexer.LexicalException;
import application.lexer.TermCode;
import application.lexer.Terminal;

/* start header from language file */

import java.util.List;
import java.util.ArrayList;

public class Parser implements SyntacticAnalyzer{

  private LexicalAnalyzer lexer;

  /* declarations from language file */

  String outEqual;

  public Parser(){
    this.lexer = new Lexer();
    __init();
  }

  /* implementations from language file */

  /* Funkcia na scitanie dvoch argumentov */
  int scitaj(int arg1, int arg2)
  {
    int i = arg1+arg2;
    return i;
  }

  /* Funkcia na odcitanie dvoch argumentov */
  int odcitaj(int arg1, int arg2)
  {
    int i = arg1-arg2;
    return i;
  }

  /* Funkcia na vynasobenie dvoch argumentov */
```

```
int vynasob(int arg1, int arg2)
{
  int i = arg1*arg2;
  return i;
}

/* Funkcia __init sa automaticky zavola pred zaciatkom parsovania */
public void __init()
{
  System.out.println("-- Vypocet hodnoty vyrazu --");
  outEqual = "=";
}



/**
 * parsing subroutine for nonterminal S
 */
private void nont_S()
throws SyntaxException, LexicalException, IOException {
  Terminal terminal = new Terminal();

  int nont_E_1;

  nont_E_1 = nont_E();
  /* semantic action body */
  {
    System.out.print(outEqual + " " + nont_E_1);
  }
}

/**
 * parsing subroutine for nonterminal E
 */
private int nont_E()
throws SyntaxException, LexicalException, IOException {
  Terminal terminal = new Terminal();
  Terminal term_plus_1, term_minus_1;

  int nont_E_0, nont_E_1new;
  int nont_E_1;
```

```
  int nont_T_1;

  nont_T_1 = nont_T();
  /* semantic action body */
  {
    nont_E_1new = nont_E_0 = nont_T_1;
  }
  lexer.lookTerm(terminal);
  while ((terminal.getTermCode() == TermCode.Term_plus)||
  (terminal.getTermCode() == TermCode.Term_minus) ) {
    nont_E_1 = nont_E_1new;
    lexer.lookTerm(terminal);
    switch (terminal.getTermCode()){
      case Term_plus:
        term_plus_1 = lexer.matchTerm(TermCode.Term_plus);
        nont_T_1 = nont_T();
        /* semantic action body */
        {
          nont_E_1new = nont_E_0 = scitaj( nont_E_1, nont_T_1);
        }
        break;
      case Term_minus:
        term_minus_1 = lexer.matchTerm(TermCode.Term_minus);
        nont_T_1 = nont_T();
        /* semantic action body */
        {
          nont_E_1new = nont_E_0 = odcitaj( nont_E_1, nont_T_1);
        }
        break;
      default:
        throw new SyntaxException("syntax_Error");
    }
    lexer.lookTerm(terminal);
  }
  return nont_E_0;
}

/**
 * parsing subroutine for nonterminal T
 */
private int nont_T()
```

```
throws SyntaxException, LexicalException, IOException {
  Terminal terminal = new Terminal();
  Terminal term_star_1;

  int nont_T_0, nont_T_1new;
  int nont_T_1;
  int nont_F_1;

  nont_F_1 = nont_F();
  /* semantic action body */
  {
    nont_T_1new = nont_T_0 = nont_F_1;
  }
  lexer.lookTerm(terminal);
  while (terminal.getTermCode() == TermCode.Term_star ) {
    nont_T_1 = nont_T_1new;
    lexer.lookTerm(terminal);
    term_star_1 = lexer.matchTerm(TermCode.Term_star);
    nont_F_1 = nont_F();
    /* semantic action body */
    {
      nont_T_1new = nont_T_0 = vynasob( nont_T_1, nont_F_1);
    }
    lexer.lookTerm(terminal);
  }
  return nont_T_0;
}

/**
 * parsing subroutine for nonterminal F
 */
private int nont_F()
throws SyntaxException, LexicalException, IOException {
  Terminal terminal = new Terminal();
  Terminal term_num_1, term_lpar_1, term_rpar_1;

  int nont_F_0;
  int nont_E_1;

  lexer.lookTerm(terminal);
  switch  (terminal.getTermCode()){
```

```
        case Term_num:
          term_num_1 = lexer.matchTerm(TermCode.Term_num);
          /* semantic action body */
          {
            nont_F_0 = Integer.parseInt( term_num_1.stringValue());
          }
          break;
        case Term_lpar:
          term_lpar_1 = lexer.matchTerm(TermCode.Term_lpar);
          nont_E_1 = nont_E();
          term_rpar_1 = lexer.matchTerm(TermCode.Term_rpar);
          /* semantic action body */
          {
            nont_F_0 = nont_E_1;
          }
          break;
        default:
          throw new SyntaxException("syntax Error");
      }
      return nont_F_0;
    }

    public void parse()
    throws SyntaxException, LexicalException, IOException{

      lexer.lexOpen("_input");
      nont_S();
      lexer.lexClose();
    }
}
```

Aby sme boli schopní odlíšiť od seba jednotlivé výskyty rovnakého symbolu v rámci jedného pravidla, má každý symbol gramatiky svoje očíslované meno v rámci metódy, ktorá reprezentuje daný neterminál.

# Kapitola 9

# Príbuzné práce

Program *JKindCons* vychádza z pôvodnej implementácie transducera *KindTran* ([6]) resp. *KindCons* ([5]). Tieto nástroje sú založené na prívetivých gramatikách.

Existuje ďalšia implementácia prívetivého transducera – program *JKind* ([9]).

Rozdiel programu *JKindCons* oproti programu *JKind* je v tom, že program *JKindCons* je implementácia konštruktora transducerov (zkonštruovaný transducer je blízky ručne napísanému). Obidve implementácie používajú hlavnú dátovú štruktúru – les produkčných stromov s výhľadmi. V prípade prívetivého transducera sa táto štruktúra používa k parsovaniu. V prípade konštruktora transducerov sa táto štruktúra používa ku generovaniu metódy, ktorá reprezentuje daný neterminál vo výslednom parseri.

Akcie sa v programe *KindTran* zapisujú pomocou sady sémantických inštrukcií. Sémantické inštrukcie v programe *JKindCons* sú prevzaté z pôvodnej implementácie z programu *KindTran*. V programe *JKindCons* boli doplnené metódy pre generovanie sémantických inštrukcií. V programe *JKind* nie je možný zápis pomocou sémantických inštrukcií.

Pôvodné nástroje sú založené na jazyku C (nástroj *KindCons* generuje parser pre jazyk C), zatiaľ čo nástroje *JKindCons* a *JKind* pracujú v Jave a preto podporujú sémantické akcie v Jave a nástroj *JKindCons* akcie v Jave aj generuje.

Výhodou programu *JKind* je, že umožňuje definovať lexikálne symboly pomocou regulárnych výrazov. Programy *KindCons* a *JKindCons* pracujú s preddefinovanými typmi terminálov – kľúčové slová, symboly, identifikátory, čísla, reťazce, znaky.

Odlišnosť programu *JKind* a *JKindCons* od programu *KindCons* je aj vo vstupnom súbore. Programy *JKind* a *JKindCons* nepotrebujú poznať obmedzenia na veľkosť rôznych štruktúr používaných v programe. Program *KindCons* tieto obmedzenia potrebuje poznať.

# Kapitola 10

# Záver

Cieľom práce bolo vytvoriť konštruktor analyzátorov a transducerov, ktorý bude pracovať s prívetivými gramatikami. Predkladaný program *JKindCons* tieto požiadavky splňuje.

Výsledný konštruktor resp. transducer pracuje s prívetivými gramatikami, čo prináša možnosť použiť v gramatike popisujúcej vstupný jazyk jednoduchú ľavú rekurziu, môžu sa vyskytovať pravidlá, ktorých pravé strany majú (netriviálny) spoločný prefix a sémantické akcie je povolené umiestňovať skoro kamkoľvek na pravú stranu pravidla – jediné miesto, kde akcie byť nemôžu, je pred ľavou rekurziou. Tieto vlastnosti umožňujú v mnohých prípadoch pohodlný zápis gramatík bez nutnosti prispôsobovať ich potrebám parseru. Ako príklad môže slúžiť gramatika aritmetických výrazov.

Program *JKindCons* pracuje s 1-prívetivými gramatikami. Generovanie zdrojových kódov parsera pre výhľad dlhší ako 1 nie je priamočiare. Vždy keď by bol použitý dlhší výhľad, korešpondujúce miesto v zdrojovom kóde by bolo ťažšie na čítanie – prinajmenšom neexistuje programová konštrukcia na rozdeľovanie viac nasledujúcich hodnôt súčastne (ako je case/switch príkaz pre jednotlivé hodnoty). Napriek tomu program počíta výhľad do dĺžky $k$ a dokáže rozhodnúť, či daná vstupná gramatika je $k$-prívetivá.

## 10.1 Ďalší možný vývoj

Tu predkladáme možnosti, ako by bolo možné aplikáciu *JKindCons* vylepšiť:

- Vylepšenia lexikálneho analyzátora:
  V súčasnosti vygenerovaný lexikálny analyzátor rozpoznáva preddefi-

nované typy terminálov - kľúčové slová, symboly, identifikátory, čísla, reťazce, znaky. Tvar niektorých je ovplivniteľný nastavením parametrov (napríklad oddeľovače reťazcov či znakov), niektoré (symboly, kľúčové slová) je možné zapísať ako literály. Možným vylepšením by bolo špecifikovať tvary terminálov pomocou regulárnych výrazov.

- Generovanie zdrojových kódov parsera pre výhľad dlhší ako 1:
  Vždy keď by bol použitý dlhší výhľad, korešpondujúce miesto v zdrojovom kóde by bolo ťažšie na čítanie – prinajmenšom neexistuje programová konštrukcia na rozdeľovanie viac nasledujúcich hodnôt súčasne (ako je case/switch príkaz pre jednotlivé hodnoty). Bolo by potrebné vyriešiť tieto problémy.

- Podpora súčasne sémantických akcií pomocou java kódu a pomocou sémantických inštrukcií. V jednom súbore s lexikálnymi a syntaktickými pravidlami, ktoré definujú jazyk sú semantické akcie definované pomocou sémantických inštrukcií a zároveň pomocou java kódu. Každá sémantická akcia (java akcia aj akcia definovaná pomocou semantických inštrukcií) by mohla mať svoj názov a v pravidle by sa používal názov tejto akcie. Takto by mohla byť použitá aj rozšírená prívetivá gramtika, to znamená oslabenie kontroly konfliktov (premenná `Extended` zo súboru so syntaktickými a lexikálnymi pravidlami), ktorá v sučastnom stave programu nie je možná pre java akcie.

# Dodatok: Obsah CD

adresár `text`

- `text.pdf` – text tejto práce.

adresár `src/build`

- `JKindCons.jar`

adresár `src/jkindcons`

- zdrojové texty programu *JKindCons* v jazyku Java, podadresáre zodpovedajú jednotlivým balíčkom, viac je možné zistiť v dokumentácii v adresáry `javadoc`.

adresár `src/javadoc` – dokumentácia k jednotlivým triedam programu *JKindCons* vo formáte HTML, hlavná stránka je index.html.

adresár `demos` – príklady vstupných súborov pre program *JKindCons*. Zároveň sa tu nachádzajú vygenerované parsery.

# Literatúra


[1] Aho A.V., Ullman J.D.: *The theory of parsing, translation, and compiling*, Volume I.:Parsing, Prentice Hall, Inc., 1972.

[2] Aho A.V.,Sethi R., Ullman J.D.: *Compilers, principles, techniques, and tools*, Addison-Wesley, 1987.

[3] Bison – http://www.gnu.org/software/bison/

[4] Linz,P.: *An introduction to Formal Languages and Automata*, Jones and Barlett Publishers, 2001.

[5] Žemlička, M.: *Kind Constructor*, 2002, http::/www.ms.mff.cuni.cz/˜zemlicka/KindCons/

[6] Žemlička, M.: *Kind Transducer*, 2002, http::/www.ms.mff.cuni.cz/˜zemlicka/KindTran/

[7] Žemlička, M.: *Principles of Kind Parsing*, Dizertačná práca, Matematicko-fyzikálna fakulta, Univerzita Karlova, Praha, 2006.

[8] Žemlička, M.: *Principles of Kind Parsing – An introduction*, Technická správa KSI MFF UK č. 2002/1, KSI MFF UK, Praha, 2002.

[9] Zvánovcová, K.: *Prívetivý translátor v Jave*, Diplomová práca, Matematicko-fyzikálna fakulta, Univerzita Karlova, Praha, 2008.

[10] Yacc – http://dinosaur.compilertools.net/yacc/index.html

[11] ANTLR – http://www.artima.com/lejava/articles/antlr_3.html

[12] Wöß A., Löberbauer M., Mössenböck H.: *LL(1) Conflict Resolution in a Recursive Descent Compiler Generator*, http://ssw.jku.at/Coco/Doc/ConflictResolvers.pdf

[13] Coco/R – http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

[14] JavaCC – https://javacc.dev.java.net/