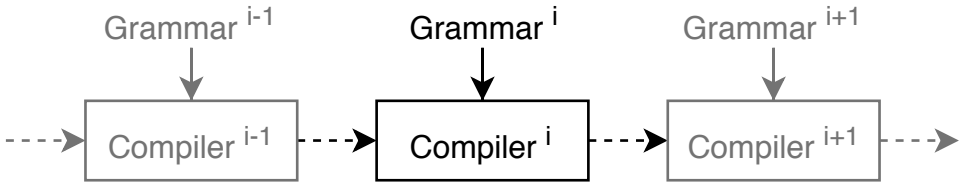
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

Generování syntaktických analyzátorů nejen regulovaných gramatik

Tomáš Dvořák*



Abstrakt

Hlavním cílem tohoto článku je představit architekturu syntaktického analyzátoru a metody jeho algoritmického generování formou nového typu překladače. Důležitým rozšířením tohoto analyzátoru je schopnost analyzovat a přijímat řetězce nepatřící do třídy bezkontextových jazyků, k čemuž byl využit hluboký zásobníkový automat. U metod algoritmického generování analyzátoru je kladen důraz na jazykovou nezávislost výstupu, čímž se výsledná architektura vyznačuje. Protože mezi vstupní gramatiky analyzátoru mohou patřit i některé gramatiky patřící mimo třídu bezkontextových gramatik, byl pro popis vstupních gramatik vytvořen speciální definiční metajazyk, který je v článku, včetně dvou příkladů, důkladně popsán. Výsledkem je prototyp překladače v jazyce Java, který realizuje překlad z definičního metajazyka gramatiky do kódu syntaktického analyzátoru, který přijímá řetězce jazyků generovaných touto gramatikou. Protože se jedná o zcela nový typ analyzátoru, je jeho funkcionalita limitována skutečně implementovanou částí rozhraní na jazyk Java. Navržený analyzátor je vhodný na větší projekty, protože umožňuje hierarchickou delegaci zodpovědnosti za zpracování jednotlivých vstupních syntaktických struktur na více tříd za využití principů dědičnosti. Samotné generování výstupního kódu je realizováno strategiemi, které se odvozují ze vstupní gramatiky.

Klíčová slova: překladač — gramatika — syntaktická analýza — generátory syntaktických analyzátorů

Přiložené materiály:

*xznebe00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Úvod

Při snaze zpracovávat strukturovaná data se jako nejefektivnější řešení jeví vlastní implementace zaměřená na konkrétní třídu dat. Se stále obsáhlejší třídou takových dat však tohle řešení nabývá na množství času vynaloženým daným programátorem. Při zpracovávání textů zdrojových kódů se jedná o desítky až stovky různých syntaktických struktur, jejichž ruční zpracování je neúnosně nákladné. Proto se v praxi využívá syntaktických analyzátorů, které na svém vstupu přijímají strukturovaná data a nějaký jejich popis, většinou ve formě gramatiky s programátorem definovaným výstupem. Tento článek rozebírá praktické možnosti dalšího ulehčení práce programátora z hlediska návrhu analyzátoru strukturovaných dat. Důležitou částí každého moderního algoritmicky vygenerovaného syntaktického analyzátoru je možnost propojení s kódem programátora, který se typicky stará o klasifikaci sémantického významu dané struktury a jejího dalšího zpracování. Toto propojení by mělo být nezávislé na konkrétní verzi vygenerovaného kódu a nemělo by vyžadovat zásah programátora do vygenerovaného kódu jakýmkoliv

způsobem.

Mezi nejvíce rozšířená existující řešení v oblasti generování syntaktických analyzátorů patří Yacc[1] a Bison[2]. Vstupem obou uvedených je soubor gramatických pravidel a výstupem vygenerovaný kód analyzátoru. Propojení s programátorským kódem poskytují formou zdrojového kódu uvnitř definičního souboru gramatiky. První z uvedených je svým výstupem limitován na jazyk C, druhý na jazyky C, C++ a Java. Dalším společným úskalím je možnost přijímat na svém vstupu pouze některé bezkontextové gramatiky. Oproti tomu výhodou je použití algoritmu LALR [1, s. 259-277] pro samotnou analýzu, která je z časového hlediska efektivní. Nespornou výhodou obou řešení je možnost využití nástroje flex[3] jako modulu lexikální analýzy.

Řešení uvedené v tomto článku vychází z myšlenky vygenerování kódu analyzátoru přes rozhraní, jehož implementaci tvoří syntaktické konstrukce jednotlivých programovacích jazyků. Pouhou implementací tohoto rozhraní je tedy možno vygenerovat analyzátor v prakticky libovolném programovacím jazyce. Z tohoto důvodu je nutno z definice vstupní gramatiky odebrat jazykově závislé syntaktické struktury propojující vygenerovaný kód s programátorským kódem a nahradit je univerzálními mechanismy. Ačkoliv se nabízí více možností, vybrány byly operace uložení tokenu na zásobník a volání uživatelské funkce.

Samotným výsledkem je generátor syntaktických analyzátorů (resp. jejich hlavních částí). Užitečnost generátoru demonstruje fakt, že výkonná část jeho syntaktického a lexikálního analyzátoru byla vygenerována kódem, který byl výstupem této operace stejného generátoru v předchozí iteraci. Uvedené řešení navíc obsahuje rozšíření ve formě podpory některých tříd gramatik, které nejsou bezkontextové. Z tohoto důvodu je doporučeno na navržený překladač nahlížet jako na prototyp, který přináší určité výhody oproti některým existujícím řešením, ale za cenu horší časové složitosti.

2. Formální základ

U čtenáře se předpokládá základní znalost teorie jazyků, syntaxí řízeného překladu a konstrukce překladačů (viz například [2] nebo [1]). V rámci uvedeného řešení struktura překladače vychází ze struktury popsané v [2]. Mezi analyzované gramatiky mohou patřit i některé gramatiky, které nepatří do třídy bezkontextových gramatik. Algoritmy navržené pro syntaktickou analýzu takových gramatik vychází z prediktivní syntaktické analýzy založené na LL(1) tabulce (viz [2]). Protože je však z praktického hlediska zbytečné navrhovat algoritmy pro každý typ gramatiky zvlášť, je zavedena obecný algoritmus regulované syntaktické analýzy, který zavádí kontext a abstrahuje metodu výběru přepisovacího pravidla.

2.1 Regulovaná syntaktická analýza

Metoda regulované syntaktické analýzy, podobně jako prediktivní syntaktická analýza, je metoda, která na svém vstupu přijme regulovanou gramatiku spolu se vstupním řetězcem a výstupem je informace o příslušnosti tohoto řetězce v dané gramatice. Vedlejším produktem je posloupnost pravidel odpovídající posloupnosti derivačních kroků z počátečního neterminálu k dané větě. V tomto případě se jedná o levý rozbor. Celý postup analýzy shrnuje algoritmus 1. Nutno podotknout, že oproti prediktivní syntaktické analýze se pracuje s hlubokým zásobníkovým automatem (viz [3]), tedy v rámci expanze se expanduje výskyt symbolu levé strany přepisovacího pravidla nejblíže vrcholu zásobníku. Při expanzi (krok 18 algoritmu 1) se před první symbol pravé strany užitého pravidla přidá pravidlo samotné, protože jeho výskytu v levém rozboru může předcházet ještě pravidlo, které se aplikuje až v následujících iteracích.

Nutno podotknout, že C je uspořádanou multimnožinou, jejíž význam určují dílčí algoritmy. Při nahrazení kroku 15 (a vypuštění kroku 16) tohoto algoritmu výběrem dat z LL(1) tabulky je metoda degradována na prediktivní syntaktickou analýzu řízenou LL(1) tabulkou. V závislosti na typu regulované vstupní gramatiky jsou nahrazeny kroky 15 a 16 algoritmu 1 dílčími algoritmy. Pravidla specifická pro výběr množiny aplikovatelných přepisovacích pravidel nazýváme *regulace*.

2.2 Maticové gramatiky

Maticovými gramatikami uvažujeme gramatiky z [4, strana 160], variantu s kontrolou na výskyt a s vymazávajícími pravidly. Pro zjednodušení uvažujme nahrazení kroku 15 a 16 algoritmu 1 funkcí, která vybere množinu pravidel M na základě následující možné derivace v souladu s definicí maticových gramatik a kontext C využije pro předepisování pravidel na základě vybraného obsahu matice. Obsahuje-li množina M více, než jedno pravidlo, odeberou se z této množiny pra-

[1] http://dinosaur.compilertools.net/
[2] https://www.gnu.org/software/bison/
[3] https://github.com/westes/flex

Algoritmus 1: Regulovaná syntaktická analýza

Vstup: Regulovaná gramatika G,
Vstupní řetězec x

Výstup: Členství řetězce x v jazyce gramatiky G,
Levý rozbor

```
1:  push(§)
2:  push(S)
3:  C ← ∅
4:  repeat
5:      X ← symbol na vrcholu zásobníku
6:      a ← aktuální symbol na vstupu
7:      if X ∈ P then
8:          output X
9:      else if X = § and a = § then
10:         Úspěch
11:     else if X ∈ Σ and X = a then
12:         Přečti další symbol a ze vstupu
13:         pop(X)
14:     else if X ∈ N then
15:         Urči množinu M přepisovacích pravidel
16:         Uprav kontext C
17:         if r = (Y, x) ∈ M and |M| = 1 then
18:             replace(Y, reversal(x) . r)
19:         else
20:             Chyba
21:     else
22:         Chyba
23: until Úspěch or Chyba
```

vidla neobsažena v prediktivní LL(1) tabulce pro danou aktuální větnou formu a symbol na vstupu. Dále uvažujme pouze takové maticové gramatiky, jejichž žádné dva řádky matice nezačínají stejným pravidlem a první prvky každého řádku matice neobsahují kontrolu na výskyt ani nejsou vymazávacím pravidlem. Regulací je v tomto případě vybraný řádek matice vázaný na první pravidlo daného řádku (dále v textu označován jako *matice*).

2.3 Programované gramatiky

Programovanými gramatikami uvažujeme gramatiky z [4, strana 163], variantu s kontrolou na výskyt a s vymazávacími pravidly. Podobně jako v předchozí kapitole uvažujme nahrazení kroku 15 a 16 algoritmu 1 funkcí, která vybere množinu pravidel M na základě následující možné derivace v souladu s definicí programovaných gramatik. Kontext C obsahuje vždy naposled využité pravidlo, na jehož základě se výběr dalších pravidel zakládá. Obsahuje-li množina M více pravidel, postupuje se stejným způsobem, jako v případě maticových gramatik. Regulací je program, tedy seznam pravidel vymezující možnosti použití pravidel následující iterace. Dále uvažujme pouze takové programované gramatiky, které pro každou větnou formu derivovatelnou z počátečního neterminálu umožní deterministický výběr jediného přepisovacího pravidla.

2.4 Gramatiky s nahodilým kontextem

Gramatikami s nahodilým kontextem uvažujeme gramatiky z [4, strana 63], variantu s kontrolou na výskyt a s vymazávacími pravidly. Podobně jako v předchozí kapitole uvažujme nahrazení kroku 15 a 16 algoritmu 1 funkcí, která vybere množinu pravidel M na základě následující možné derivace v souladu s definicí gramatik s nahodilým kontextem. Kontext C zde nenachází uplatnění. Obsahuje-li množina M více pravidel, postupuje se stejným způsobem, jako v případě maticových gramatik. Regulací je seznam neterminálů, jejichž přítomnost (nebo nepřítomnost) je nezbytná pro výběr pravidla, u kterého je uveden (dále v textu je tento seznam označován jako *kontext*). Dále uvažujme pouze takové gramatiky s nahodilým kontextem, které pro každou větnou formu derivovatelnou z počátečního neterminálu umožní deterministický výběr jediného přepisovacího pravidla.

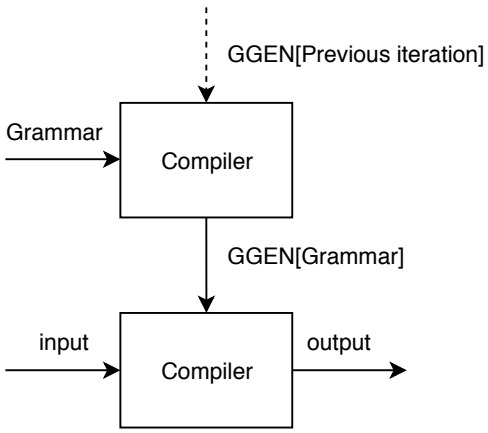
3. Architektura navrženého překladače

Myšlenku obecného principu překladače znázorňuje obrázek 1, kde na vstupu je gramatika (*Grammar*) popisující vstupní data (*input*), samotná vstupní data (*input*) a jejich překlad do uživatelských akcí (*output*). Za povšimnutí stojí, že blok *Compiler* je na obrázku dvakrát, přičemž horní výskyt je vygenerován daty předchozí iterace. Důkladnějšímu rozboru tohoto bloku je věnována pozornost v kapitole 5.

3.1 Architektura lexikálního analyzátoru

Lexikální analyzátor (pojmenovaný jako *LPM Scanner*) oproti analyzátorům, které generuje nástroj typu flex[4] neobsahuje data o lexikálních jednotkách, nýbrž tato data dostává jako parametr inicializace, a následně sestaví vnitřní struktury nutné pro analýzu.

[4] https://github.com/westes/flex



Obrázek 1: Princip funkce navrženého překladače

3.2 Architektura syntaktického analyzátoru

Jak již bylo v úvodu naznačeno, navržený překladač dokáže pracovat s více typy gramatik, na což je algoritmus 1 speciálně navržen. Ve výsledném kódu je obsažena implementace tohoto algoritmu, která je, na základě typu přijímané gramatiky, doplněna o příslušnou funkci výběru přepisovacího pravidla a úpravy kontextu. Podobně jako lexikální analyzátor je i syntaktický analyzátor navržen tak, aby neobsahoval data konkrétní gramatiky, ale tato data mu byla předána jako parametr inicializace.

4. Definiční metajazyk

Pro důkladný popis vstupní gramatiky je nutno zavést speciální jazyk (*definiční metajazyk*). Tento jazyk nesmí obsahovat propojení výsledného kódu s naprogramovanou částí analyzátoru na úrovni zdrojového kódu, jak je tomu například v některých existujících řešeních. Základními strukturami tohoto jazyka je terminál a neterminál. Terminálem se stane každá posloupnost symbolů, která odpovídá standardní definici proměnné pro jazyk C (viz [5]), nebo právě jeden tisknutelný znak. Každý neterminál je terminálem uzavřeným v ostrých závorkách (například `<S>`). Pro zápis prázdného řetězce (ϵ) je použit symbol `e`, což však nijak nezabraňuje výskytu symbolu `e` jako terminálu (jen se jeho zápis musí označit jinak). Každá metainformace popisující vlastnosti překladu se nachází na začátku řádku a je uvozena znakem `@`. Tyto metainformace označujeme jako *anotace*. Přehled dostupných anotací a dalších jazykových konstrukcí je předmětem následujících podkapitol. Řádkové a blokové komentáře mají stejný zápis, jako v jazyce C (viz [5]). Nejdůležitějším požadavkem je však příslušnost zápisu definice gramatiky popisující metajazyk v jazyce, který sama popisuje.

4.1 Zápis pravidel

Zápis pravidel představuje nejdůležitější syntaktickou strukturu definičního metajazyka. Každé pravidlo musí začínat na novém řádku, a to volitelnou regulací. Přehled podporovaných regulací nabízí obsah tabulky 1.

Tabulka 1: Tabulka regulací

Název regulace	Příklad zápisu
Matice	`$1, 1, 2+: <A>->a` `$2: <B >->b`
Program	`$1, $1 | $2: <A> -> a` `$2: <B>->b`
Kontext	`$1,(<B>)|(<C>):<A>->a`

Zápis každé regulace začíná označením pravidla prefixem `$`, tělem regulace a končí dvojtečkou, za kterou následuje samotné pravidlo. Podtržená část regulace označuje identifikátor pravidla. Pro zápis maticové regulace je tělem regulace řádek matice, který začíná pravidlem, u něhož se regulace nachází (i toto je potřeba zapsat). Pro zápis programové regulace je tělem regulace seznam pravidel úspěchu a volitelně za oddělovačem také seznam pravidel neúspěchu. Programová regulace je od maticové rozlišena prefixy `$` před identifikátory pravidel. Pro kontextové regulace je tělem regulace uzávorkovaný seznam povolujících a volitelně za oddělovačem i omezujících neterminálů. V rámci jednoho definičního souboru lze použít nejvýše jeden typ regulace na všechna pravidla (nelze např. kombinovat maticové a programované gramatiky).

4.2 Anotace

Anotace obsahují metainformace upravující požadované chování překladače. Jejich přehled je v tabulce 2. Tučně vyznačené anotace jsou pro každou definici gramatiky povinné. Anotace `@Name` umožňuje přejmenovat terminál v rámci dané definice, a současně změnit i jeho název ve vygenerovaném kódu, což je užitečné pro pojmenování struktur, které nemohou být součástí syntaxe nebo sémantiky výstupního kódu, ale mohou se vyskytovat v definičním metajazyce (např. definice operátorů metajazyka v definici metajazyka). Anotace `@Ignore` udává pomocí regulárních výrazů struktury, které se mohou objevit na vstupu, ale jejich výskyt je ignorován. Praktickým příkladem užití této anotace mohou být bílé znaky a komentáře, které zpravidla nebývají součástí syntaxe moderních programovacích jazyků.

Tabulka 2: Tabulka anotací

Anotace	Parametry	Význam
`@Main`	Neterminál	Označení počátečního neterminálu
`@Stack`	Terminál	Deklarace uživatelského zásobníku
`@Func`	Terminál	Deklarace uživatelské funkce
`@Name`	2 terminál	Přejmenování terminálu
`@Lex`	Terminál, Řetězec	Definice lexému regulárním výrazem
`@Include`	Řetězec	Vložení souboru
`@IncludeOnce`	Řetězec	Unikátní vložení souboru
`@Ignore`	Řetězec	Regulární výraz bílých znaků
`@EndOfFile`	Terminál	Specifikace terminálu konce vstupu

4.3 Kvantifikátory

Definiční metajazyk dovoluje užití kvantifikátorů pro zkrácení zápisu pravidel. Užitím kvantifikátoru však vznikají další neterminály a pravidla, což nemusí být vždy žádoucí s ohledem na vlastnosti regulovaného překladu. Přehled použitelných kvantifikátorů shrnuje tabulka 3.

Tabulka 3: Tabulka kvantifikátorů

Zápis	Význam	Překlad do pravidel

`( )`	Sjednocení úseku pravé části pravidla	`<A>->(a b c)`
		`<A>-><A_2>`
		`<A_2>->a b c`

`*`	Libovolný počet výskytů	`<A>->a*`
		`<A>-><A_star>`
		`<A_star>->e`
		`<A_star>->a<A_star>`

`+`	Libovolný nenulový počet výskytů	`<A>->a+`
		`<A>-><A_plus>`
		`<A_plus>->a<A_star>`

`?`	Volitelný výskyt	`<A>->a?`
		`<A>-><A_opt>`
		`<A_opt>->e`
		`<A_opt>->a`

`|`	Alternativy	`<A>->a|b`
		`<A>->a`
		`<A>->b`

Význam posledního sloupce tabulky 3 spočívá v praktickém znázornění důsledků užití daného kvantifikátoru. Pravidlo, které je podtrženo, vyjadřuje zápis v definičním metajazyce, avšak místo tohoto pravidla se stanou součástí gramatiky pravidla nepodtržená, realizující daný kvantifikátor. Takto vzniklá pravidla označujeme jako *syntetická pravidla* a každý nově vzniklý neterminál jako *syntetický neterminál*. Kvantifikátory je možno téměř libovolně sdružovat a zanořovat, pokud výsledná gramatika zůstane LL(1) gramatikou (nebo bude splňovat podmínky kladené na daný typ regulované gramatiky).

4.4 Práce s uživatelskými objekty

K propojení vygenerované části analyzátoru s programátorským kódem uživatele je zapotřebí zavést objekty nezávislé na daném programovacím jazyce. Vybranými objekty jsou zásobník a funkce, neboť jsou oba reprezentovatelné prakticky ve všech moderních programovacích jazycích.

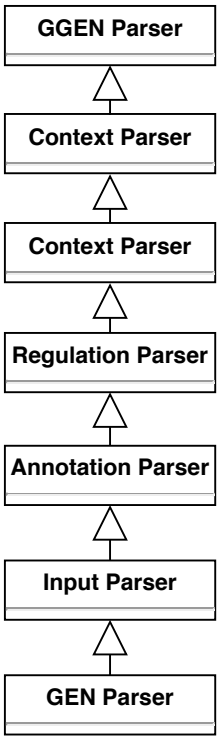
Nedílnou součástí syntaxe definičního metajazyka je tudíž podpora práce s těmito objekty. Každému užití uživatelského objektu musí předcházet jeho deklarace. Pro uživatelské zásobníky je definována operace uložení daného tokenu na zásobník (např. zápis `a => s1 => s2 => s3` způsobí uložení tokenu *a* do zásobníků *s1*, *s2* a *s3*). Zápisy uložení na zásobník mohou být pouze na pravé straně přepisovacích pravidel za terminály. Pro uživatelské funkce je definovaná operace volání dané funkce (např. zápis `<a> $f1 $f2 $f3` způsobí zavolání funkcí *f1*, *f2* a *f3* po redukci celého produktu neterminálu `<a>`). Volání uživatelské funkce může být pouze na pravé straně přepisovacích pravidel, a to za terminály i neterminály. Pořadí volání funkcí uvedených za sebou je nedefinované. V případě operací se zásobníky a volání funkcí v rámci stejného neterminálu se provedou nejdříve operace nad zásobníky, a poté až volání funkcí bez ohledu na pořadí zápisu.

5. Implementace a možnosti využití

Oblast využití je díky navržené architektuře velmi rozsáhlá. Pro názornost je zde popsáno využití uvedeného generátoru pro implementaci překladače definičního metajazyka v programovacím jazyce Java.

Při návrhu bylo využito předchozí iterace generátoru pro vygenerování syntaktického a lexikálního analyzátoru v daném jazyce. Na obrázku 3 je znázorněna struktura implementovaného překladače, přičemž právě vyobrazené třídy *LPM Scanner* a *GEX Parser* představují výstup jeho předchozí iterace (v rámci implementace se však z důvodu vyšší efektivity negenerují, neboť jsou v Javě pro všechny podporované gramatiky neměnné). Vstup tvoří řetězec *input* třídy *GEX Parser* a výstup tohoto bloku vyjadřuje příslušnost tohoto vstupního řetězce do jazyka gramatiky, na

základě které byla třídě *GEX Parser* vygenerována.

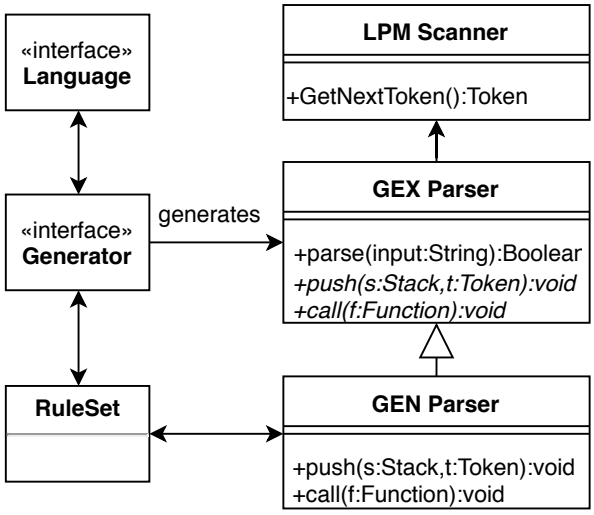


Obrázek 2: Diagramu tříd navrženého parseru

Jádro implementace je skryto ve třídě *GEN Parser*, která implementuje operace nad uživatelskými objekty (nebo jejich implementace dědí). Pokud byl vstupní řetězec *input* úspěšně přijat třídou *GEX Parser*, je vedlejším výstupem instance třídy *RuleSet*, která obsahuje vnitřní reprezentaci vstupní gramatiky.

Následně je na základě dodané implementace rozhraní *Generator* (strategie generování výstupního kódu) a *Language* (rozhraní popisující syntaktické konstrukce výstupního jazyka) vygenerován zdrojový kód ve výstupním jazyce. Jako implementace rozhraní *Generator* může sloužit například popis algoritmu prediktivní syntaktické analýzy pro bezkontextové jazyky, nebo v tomto článku uvedený algoritmus regulované syntaktické analýzy. Uvedená implementace rozšiřuje překladač o strategii výstupu ve formě binární reprezentace vstupního řetězce pro snadné využití v projektech pracujících s gramatikami zadávanými za běhu.

Konkrétní význam třídy *GEN Parser* z obrázku 3 tvořící jádro syntaktické analýzy je znázorněn na obrázku 2. Pro jednoduchost byly některé vztahy a obsahy jednotlivých tříd vypuštěny. Obsah třídy *GGEN Parser* je rovněž vygenerován, a obsahuje definice uživatelských objektů (viz 4.4) a vstupní data lexikální a syntaktické analýzy (inicializační data tříd *LPM Scanner* a *GEX Parser*). Zodpovědnost za implementaci uživatelských funkcí vygenerovaného kódu (a tedy propojení s programátorským kódem) je delegována dědičností. Například syntaktická struktura definičního metajazyka regulace se vyhodnotí uvnitř třídy *Regulation parser*, pravidlo gramatiky uvnitř třídy *Rule parser* atd.



Obrázek 3: Diagram tříd navrženého překladače

5.1 Možnosti využití

Při vytváření nových analyzátorů je nezbytné definovat gramatiku (a přeložit ji na třídu *GGEN Parser*) a vytvořit strukturu tříd podobnou obrázku 2 (v hierarchii níže než *GGEN Parser*) definující uživatelské funkce deklarované v souboru gramatiky. Dále je nutno přidat reference na třídy *GEX Parser* a *LPM Scanner* pro úspěšné spuštění kódu (některá vývojová prostředí zajistí automaticky). Při vytvoření instance třídy dědící z třídy *GGEN Parser* a zavolání funkce *GEX Parser:parse* (která je příhodně dostupná i z třídy uživatelských parserů) je provedena lexikální i syntaktická analýza, v rámci které jsou provedeny operace nad uživatelskými objekty pro zadaný vstupní řetězec.

5.2 Příklady

Následují dva příklady demonstrující zápis vybraných syntaktických struktur smyšleného programovacího jazyka a jeho sémantiku v definičním metajazyce.

První příklad ukazuje zpracování deklarace proměnné. Syntaxi lze vyjádřit jako zápis datového typu následovaný názvem proměnné a středníkem. Propojení s programátorským kódem by v tomto případě probíhalo na základě volání funkce `HaveVar`, která je ve vygenerovaném kódu deklarovaná jako abstraktní.

```
1  @Main <vardef>
2  @EndOfFile EOF
3  @Ignore "[ \t\n\r]"
4  @Stack Type
5  @Stack VarName
6  @Func HaveVar
7  <vardef> -> <type> name => VarName ;
       $HaveVar
8  <type> -> (unsigned => Type)? (int
       =>Type | float =>Type)
9  <type> -> String => Type
10 @Lex name "[a-zA-Z][a-zA-Z0-9]*"
```

Příklad 1: Ukázka definice deklarace proměnné

Analyzátor takového programovacího jazyka by měl podobnou architekturu, jako je ta na obrázku 2. Uvnitř funkce `HaveVar` by četl obsah zásobníků `Type` a `VarName`, které by obsahovaly po řadě datový typ a název deklarované proměnné.

Druhý příklad, oproti tomu předcházejícímu, umožňuje díky využití maticové gramatiky přijímat syntaktickou strukturu $type^n\, name^n\, value^n$, která není bezkontextová. Metoda regulované syntaktické analýzy (viz algoritmus 1) navíc zajistí, že v době volání uživatelské funkce `HaveVar` obsahují poslední symboly zásobníků `Type`, `VarName`, `Value` po řadě typ, název a hodnotu proměnné, jakoby se jednalo o sekvenční definici $(type\, name\, value)^n$.

```
1  @Main <var_def>
2  @EndOfFile EOF
3  @Ignore "[ \t\n\r]"
4  @Stack Type
5  @Stack Name
6  @Stack Value
7  @Func HaveVar
8  $1,1: <var_def> -> <type> <A> <name>
       <B> <value>
9  $2,2,3: <A> -> <type> <A>
10 $3: <B> -> <name> <B> <value>
11 $4,4,5: <A> -> e
12 $5: <B> -> e
13 $6,6: <type> -> Type => Type
14 $7,7: <name> -> Name => Name
15 $8,8: <value> -> Value => Value
       $HaveVar
16 @Lex Type "(int|string)"
17 @Lex Name "[a-zA-Z_][a-zA-Z0-9_]*"
18 @Lex Value "(\"[a-z]*\")|([0-9]+)"
```

Příklad 2: Ukázka využití regulované gramatiky

6. Závěr

Článek pojednával o návrhu překladače schopného generovat části jiných překladačů nezávislých na konkrétním programovacím jazyce a jeho konkrétní implementaci v jazyce Java, která využívá výstup svých předchozích iterací jako modulů lexikální a syntaktické analýzy. V článku je rovněž zahrnuta kapitola zabývající se praktickému využití překladače pro tvorbu částí jiných překladačů.

Ačkoliv se jedná pouze o prototyp, oproti některým podobným projektům poskytuje vylepšení ve formě možnosti pracovat s některými regulovanými gramatikami a přijímat řetězce patřící do třídy jazyků, které nejsou bezkontextové.

Přestože je prototyp vázán na programovací jazyk Java, implementací rozhraní jazyka je možno vygenerovat část kódu lexikálního i syntaktického analyzátoru překladače v libovolném jiném programovacím jazyce.

Navržený překladač disponuje vlastností algoritmicky generovat analyzátory strukturovaných dat, což nachází uplatnění všude tam, kde taková analýza probíhá. Samotný prototyp překladače je aktuálně využíván jako součást experimentálního interpretu hybridního programovacího jazyka a jeho vývoj se bude dále ubírat směrem k optimalizovanému a efektivnímu překladu.

Poděkování

Tímto bych chtěl poděkovat vedoucímu své diplomové práce, prof. RNDr. Alexanderu Medunovi, CSc, za jeho čas a užitečné myšlenky. Dále bych chtěl poděkovat Dennisovi Ÿingovi, za jeho inovativní nápady.

Literatura

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques and tools*. Addison-Wesley, 1985.

[2] Alexander Meduna. *Elements of compiler design*. Auerbach, 2008. ISBN: 978-1-4200-6323-3.

[3] Alexander Meduna. Deep pushdown automata. *Acta Informatica*, 2006(98):114–124, 2006.

[4] Alexander Meduna and Petr Zemek. *Regulated Grammars and Automata*. Springer New York, 2014. ISBN: 978-1-4939-0368-9.

[5] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.