Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Možnosti rozširovania syntaxe programovacieho jazyka Java

2012 **Viktor Dobránsky**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky
Katedra počítačov a informatiky

Možnosti rozširovania syntaxe programovacieho jazyka Java

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	prof. Ing. Ján Kollár, CSc.
Konzultant:	Ing. Sergej Chodarev

Košice 2012 **Viktor Dobránsky**

Analytický list

Autor:	Viktor Dobránsky
Názov práce:	Možnosti rozširovania syntaxe programovacieho jazyka Java
Podnázov práce:	
Jazyk práce:	slovenský
Typ práce:	Diplomová práca
Počet strán:	**??**
Akademický titul:	Inžinier
Univerzita:	Technická univerzita v Košiciach
Fakulta:	Fakulta elektrotechniky a informatiky (FEI)
Katedra:	Katedra počítačov a informatiky (KPI)
Študijný odbor:	9.2.1 Informatika
Študijný program:	Informatika
Mesto:	Košice
Vedúci práce:	prof. Ing. Ján Kollár, CSc.
Konzultanti práce:	Ing. Sergej Chodarev
Dátum odovzdania:	27. 4. 2012
Dátum obhajoby:	24. 5. 2012
Kľúčové slová:	syntax, Java, rozširovanie, AHEAD
Kategória konspekt:	Výpočtová technika
Citovanie práce:	Viktor Dobránsky: Možnosti rozširovania syntaxe programovacieho jazyka Java. Diplomová práca. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky. 2012. **??** s.
Názov práce v AJ:	Possibilities to extend Syntax of Java Programming Language
Podnázov práce v AJ:	
Kľúčové slová v AJ:	extend, AHEAD, Java, syntax

Abstrakt v SJ

Táto diplomová práca sa zaoberá nástrojmi schopnými rozšíriť jazyk *Java* novými konštrukciami. Medzi ne patria nástroje ako sú *Ahead Tool Suite*, *Java Syntactic Extender*, *Maya*, *SugarJ* a *Project Lombok*. Jeden z nich bol vybraný a použitý na vytvorenie jazykového rozšírenia. Na základe ich porovnania bol vybraný nástroj *Ahead Tool Suite*. Rozšírenie syntaxe jazyka *Java* vychádza z nástroja *LINQ*. *LINQ* je rozšírením jazyka *C#*. Umožňuje písať *SQL* príkazy na selekciu dát priamo do zdrojového kódu programu. Konečným výsledkom diplomovej práce je program, ktorý je schopný preložiť zdrojový kód s definovaným rozšírením do zdrojového kódu v čistej *Jave*. Výstupný zdrojový kód môže byť následne preložený *Java* kompilátorom. Tiež bolo pridané nové kľúčové slovo `var`, ktoré sa píše namiesto dátového typu. Toto kľúčové slovo je použité ako informácia pre prekladač aby našiel výsledný dátový typ.

Abstrakt v AJ

This thesis deals with tools which are able to extend the *Java* language with new structures. These include tools such as *Ahead Tool Suite*, *Java Syntactic Extender*, *Maya*, *SugarJ* and *Project Lombok*. One tool has been selected and it is used to create language extension. Based on their comparison tool *Ahead Tool Suite* was selected. The extension of syntax of *Java* language is based on tool *LINQ*. *LINQ* is a language extension for *C#* language. It enables an ability to write *SQL* statements for data selection directly into the source code of the program. The final result of this thesis is a program which translates the source code with the extension into the source code in pure *Java* language. The output of the source code can be translated by the *Java* compiler. Keyword `var` was added and used instead of data type. This keyword is used as an information for the translator to find the resulting data type.

Čestné vyhlásenie

Vyhlasujem, že som diplomovú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice 27. 4. 2012 .....

Vlastnoručný podpis

Poďakovanie

Ďakujem vedúcemu diplomovej práce prof. Ing. Jánovi Kollárovi, CSc. za poskytnutie tejto diplomovej témy, ktorá mi pomohla rozšíriť moje poznatky v obľúbenej oblasti informatiky.

Tiež by som sa chcel poďakovať môjmu konzultantovi diplomovej práce Ing. Sergejovi Chodarevovi za cenné rady, pripomienky a trpezlivosť.

V neposlednom rade chcem poďakovať svojej rodine a priateľom za podporu počas štúdia na vysokej škole.

Predhovor

Táto práca je charakteristická svojim obsahom, ktorý sa venuje nástrojom umožňujúcim rozširovanie jazyka *Java* a samotnému rozšíreniu tohto jazyka. Obsahuje veľký počet ukážok zdrojových textov, ktoré majú pomôcť čitateľovi lepšie pochopiť zmysel textu. Spracovanie práce trvalo v rozsahu troch semestrov. Hlavným dôvodom výberu tejto témy bolo zdokonalenie svojich schopností v programovacom jazyku *Java*. Myšlienka rozširovania syntaxe programovacieho jazyka je veľmi zaujímavá a do značnej miery môže uľahčiť ďalšiu prácu s týmto jazykom. Konečným výsledkom je program, ktorý zabezpečuje prevod zdrojového kódu obsahujúceho rozšírenie do odpovedajúceho zdrojového kódu v čistom jazyku *Java*. Práca slúži ako ukážka postupu pri tvorbe rozšírenia a jeho aplikovaní do programovacieho jazyka. Pri práci bolo použité objektové programovanie a programovací jazyk *Jakarta*, ktorý je rozšírením programovacieho jazyka *Java*. Na definovanie syntaxe nového rozšírenia sa používal jazyk *Bali*. Na uchovanie prenášaných údajov sa používal sémantický model.

Obsah

Zoznam obrázkov

Slovník termínov

Ant je *Java* knižnica a nástroj, ktorý umožňuje riadiť procesy opísané v stavebných súboroch.

APT z angl. *Annotation Processing Tool* je nástroj pre spracovanie anotácií.

AST z angl. *Abstract Syntax Tree* je stromová reprezentácia abstraktnej syntaktickej štuktúry zdrojového kódu napísaného v programovacom jazyku.

BNF z angl. *Backus-Naur Form* je používaná k vyjadreniu bezkontextovej gramatiky. Používa sa na popis formálnych jazykov.

JAR z angl. *Java ARchive* je archivačný súborový formát typicky použitý na agregáciu *Java class* súborov.

JDBC z angl. *Java DataBase Connectivity* je rozhranie poskytujúce jednotný prístup k databázam.

IDE z angl. *Integrated Development Environment* je integrované vývojové prostredie pre programovacie jazyky.

JDO z angl. *Java Data Objects* definuje rozhranie perzistentných *POJO* v úložisku dát.

JPA z angl. *Java Persistence API* je *Java* programovací jazykový framework slúžiaci na manažment relačných dát v aplikáciách.

POJO z angl. *Plain Old Java Object* je zdôraznenie obyčajného *Java* objektu. Používa sa na označenie *Java* objektov, ktoré nepoužívajú žiadne z hlavných *Java* objektových modelov, konvencií alebo frameworkov.

Úvod

Prioritou tejto práce je ukázať ako je možné rozšíriť syntax jazyka *Java*. Rozširovaním syntaxe jazyka sa rozumie pridanie nových možností písania zdrojových kódov k už existujúcim možnostiam jazyka. Na rozširovanie syntaxe jazyka existujú rôzne nástroje, ktoré sú v práci spracované. Ako inšpirácia rozšírenia poslúžil nástroj *LINQ* od spoločnosti *Microsoft* a nástroje jemu podobné pre prostredie *Java*. Nástroj *LINQ* ponúka možnosť písať dopyty ako súčasť zdrojového kódu. Syntax takéhoto dopytu je pritom podobná jazyku *SQL* a obsahuje rovnaké kľúčové slová. Jazyk *SQL* slúži na výber, vkladanie, mazanie a úpravu dát. V tomto prípade sú dáta iba čítané. Problém nástrojov pre jazyk *Java* umožňujúcich písať dopyty priamo v zdrojovom kóde je ten, že syntax týchto dopytov nie je veľmi podobná syntaxi jazyka *SQL*. To potom spôsobuje horšiu čitateľnosť týchto príkazov.

Prvá kapitola sa venuje nástrojom, ktoré umožňujú rozšíriť syntax programovacieho jazyka *Java*. V závere tejto časti sú nástroje porovnané a na základe tohto porovnania je vybraný jeden nástroj pomocou ktorého sa vytvorí ukážkové rozšírenie jazyka *Java*.

Druhá kapitola sa venuje rozšíreniu programovacieho jazyka *C#* s názvom *LINQ*. V tejto kapitole sú stručne opísané aj nástroje pre prostredie *Java*, ktoré sú svojou funkcionalitou podobné nástroju *LINQ*.

V tretej časti diplomovej práce je navrhnuté rozšírenie pre jazyk *Java*. Navrhnutá je jeho syntax, rozbalenie do čistej *Javy* a štruktúra riešenia.

Štvrtá kapitola sa venuje samotnej implementácií definovného rozšírenia jazyka *Java*. Sú v nej načrtnuté riešenia niektorých problémov s ukážkami zdrojových textov.

1 Nástroje umožňujúce rozšíriť syntax programovacieho jazyka Java

Jazyk nie je nič iné ako množina reťazcov nad abecedou. Na opísanie konkrétnej syntaxe programovacích jazykov sa používajú gramatiky. Konkrétna syntax programovacích jazykov je väčšinou definovaná bezkontextovou gramatikou. Jazyky sú klasifikované podľa expresivity gramatík, ktoré ich opisujú [16].

Úlohou syntaktickej analýzy je určiť, či daný reťazec patrí jazyku alebo nie. Ak je reťazec v danom jazyku, syntaktický analyzátor musí o tom vrátiť nejaký dôkaz. Dôkazom môže byť napríklad syntaktický strom. Syntaktická analýza spája konkrétnu a abstraktnú syntax [16].

Abstraktná syntax hovorí o štruktúre programov v danom jazyku a nie o samotných reťazcoch. Zaujíma sa o štruktúru elementov a ich generovanie. Na jazyk sa teda díva ako na induktívnu definíciu [16].

V gramatikách sa používajú syntaktické kategórie ako výrazy a príkazy pre vyjadrenie formovania viet programovacieho jazyka. Tieto zápisy nie sú však úplne dobré na precíznu definíciu výrazov. Výrazy sú preto ďalej rozdelené do rozličných typov. Typový systém definuje ako môžu byť tvorené výrazy [16].

Účelom sémantík programovacích jazykov je vytvorenie významu pre každý syntaktický správny výraz v programovacom jazyku. Definícia sémantiky programovacích jazykov je základom pre prekladače a kompilátory jazykov [16].

Rozširovanie programovacieho jazyka o nové konštrukcie je výhodné v prípade, ak existuje predpoklad, že sa rozšírenie bude používať často. Cieľom rozširovania syntaxe programovacieho jazyka je uľahčenie programátorskej práce. Snahou je zvýšiť abstrakciu a znížiť špecializáciu programovacieho jazyka.

Medzi nástroje schopné rozšíriť syntax programovacieho jazyka *Java* je možné radiť

nástroje ako sú *Ahead Tool Suite*, *Lombok*, *SugarJ*, *Java syntactic extender* a *Maya*. Každý zo spomínaných nástrojov umožňuje rozširovanie syntaxe iným spôsobom.

1.1 Java syntactic extender

Tento nástroj umožňuje, ako z názvu vyplýva rozšíriť syntax jazyka *Java*. Na vývoji sa podieľali *Jonathan Bachrach* a *Keith Playford*. Nástroj vyžaduje určité zlepšenia o ktorých autori hovoria na svojej stránke. Vývoj tohto nástroja sa zastavil v roku 2003. Inšpirovaním pri tvorbe *JSE* bol makro systém jazyka *Dylan* a jeho hlavnou myšlienkou bolo vytvoriť flexibilný makro systém, ktorý by bol zároveň aj veľmi silným nástrojom a umožnil zjednodušené písanie *Java* kódu [13].

JSE predstavuje syntaktický preprocesor, ktorý spolupracuje s kódom cieľového jazyka. Pre manipuláciu s fragmentmi zdrojového kódu *JSE* používa kostrové syntaktické stromy (z angl. skeleton syntax tree - *SST*). *SST* pracujú na abstraktnej syntaxi jazyka a obsahujú iba najužitočnejšie tvary pre spracovanie makier. *SST* môže byť vytvorený manuálne, alebo môže byť vytvorený použitím konkrétnej syntaxe cez kódové úvodzovky (z anlg. *code quotes*).

Technika porovnávania vzorov (z angl. *pattern matching*) je použitá na pridávanie nových syntaktických foriem do *Java* gramatiky. *SST* štruktúry špecifikované ako porovnávacie vzory a identifikátory sú zviazané so špecifickými časťami stromu pre použitie v asociovanom preklade. Pravidlá pre špecifické konštrukcie sú skúmané metódou zhora nadol, dokiaľ nie je nájdený odpovedajúci vzor. Ak existujú viacero možných odpovedajúcich vzorov, tak sa použije prvý z nich. Neexistuje tu žiadne zabezpečenie pre nejednoznačnosti zavedené do gramatiky. Keď zdrojový kód napísaný v rozšírenej syntaxi je konvertovaný do štandardného *Java* kódu, makro rozširovač (z angl. macro expander) pracuje hore - dole, rozširujúc makrá podľa pravidiel dokiaľ žiadne neostanú. Pre každý identifikátor, ktorý môže byť rozšírený makrom, *JSE* hľadá `class` súbor s preddefinovaným menom a pokúša sa ho rozšíriť. Z toho

vyplýva, že všetky makrá musia začínať s požadovaným menom [30].

Makro definuje syntaktické rozšírenie v nejakom základnom jazyku a dovoľuje používateľom definovať význam jednej konštrukcie v pojmoch iných konštrukcií. Tieto deklarácie sú volané makro definície a ich použitie je nazývané makro volanie. Makro poskytuje silnú abstrakciu a to vďaka jasnosti, stručnosti a skrývaní implementácie. Makro volania sú limitované pre niektoré kontexty a tvary zodpovedajúce súčasným *Java* syntaktickým kontextom a tvarom. Tvary poskytujú ľahké nájdenie pozície konca makra pred odovzdaním do makro rozširovača [13].

Syntaktické rozširovače sú triedy, ktoré implementujú rozhranie `SyntaxExpander` a nasledujú priradenú menovú konvenciu. Tieto triedy koexistujú so spustiteľnými súbormi *Javy* a so `class` súbormi programu a na požiadanie sú dynamicky načítané do kompilátora alebo preprocesora v čase kompilácie. Syntaktické rozširovače môžu byť definované na najvyššej úrovni, alebo prežívať vnútri tried [13].

Implementovanie rozširovacieho nástroja je jednoduché. Podľa návodu stačí iba pridať *JSE* knižnicu do *Java* projektu a mierne upraviť spúšťací *ANT* súbor. V ňom je potrebné len prepojiť makrá, ktoré majú koncovú príponu `jse` s *Java* súbormi.

Zdrojový kód makra je možno rozdeliť na dve časti. Prvou je definovanie syntaxe a teda zápisu pridávaného kódu, druhou je definovanie sémantiky a teda významu rozšírenia. Tu je možné určiť ako sa rozbalí dané rozšírenie do podoby vhodnej pre spracovanie *Java* kompilátorom. Príklad zdrojového kódu makra je uvedený na obrázku 1 – 1 [13].

Tento príklad je asi najjednoduchšia ukážka makra. V podstate sa tu jedná o obohatenie jazyka *Java* s kľúčovým slovom `when`. Toto doplnenie má presne takú istú funkciu ako klauzula `if` a teda očakáva ako parameter pravdivostnú hodnotu. Pričom ak je táto hodnota pravdivá vykonajú sa príkazy v bloku, ak nie je pravdivá tak sa pokračuje vynechaním nasledujúceho bloku príkazov. Je možné si všimnúť rozdelenie na sémantickú časť a syntaktickú časť. Za príkazom `case` je definovaná

```
public syntax when {
    case #{
        when (?exp:expression)
            ?stmt:statement
        }:
    return #{
        if (?exp) {
            ?stmt
        }
    };
}
```

Obrázok 1 – 1 *JSE* makro definícia [13]

syntax a za príkazom `return` je definovaná sémantika nového rozšírenia. Príkaz `case` je možné chápať ako zovšeobecnenie príkazov `switch/case` v *Jave* s určením pre porovnávacie vzory [13].

Ďalší príklad uvádza riešenie konfliktu identifikátorov premenných. V prípade, ak pri generovaní nového zdrojového kódu bude generovaná aj premenná s rovnakým identifikátorom aký už existuje, tak je potrebné aplikovať na zdrojový kód tzv. hygienu. Ak je hygiena realizovaná v *JSE* rozšírení, tak to znamená, že preprocesor si zoberie na starosť tieto konflikty a opatrne premenuje všetky lokálne premenné. Pri generovaní premennej do nového rozšírenia je preto potrebné použiť metódu `genSym` s parametrom generovaného symbolu, aplikovanú na triedu `IdentifierFragment`. Príkladom je makro obohacujúce jazyk o rozšírenie nového cyklu, ktoré je možné vidieť na obrázku 1 – 2 [13].

Ďalší príklad 1 – 3 ilustruje použitie viacnásobného párovania vzorov. Rozšírenie s názvom `assert` je schopné akceptovať jeden alebo dva argumenty. Jeho funkcia spočíva v zistení či je argument pravdivý a ak áno, tak vyvolá výnimku. V prípade, ak je nastavená hodnota aj druhého argumentu, tak táto výnimka bude obsahovať text uvedený v druhom argumente príkazu [13].

```
public syntax foreach {
    case #{
        foreach (?t:type ?element:name in ?exp:expression)
            ?stmt:statement
    }:{
        Fragment i = IdentifierFragment.genSym(„i");
        return #{
            for(java.util.Iterator ?i = ?exp.iterator(); ?i.hasNext(); ) {
                ?t ?element = (?t) ?i.next();
                ?stmt
            }
        };
    }
}
```

Obrázok 1 – 2 *JSE* makro - nový cyklus [13]

Pri tvorbe rozšírenia nemusia byť vždy pridávané nové kľúčové slová, ale je možné rozšíriť význam už existujúcich kľúčových konštrukcií jazyka *Java*. To ilustruje príklad uvedený na obrázku 1 – 4. Toto makro zabezpečí automatické vytvorenie zapuzdrenia pre privátne premenné triedy. Vytvoria sa teda metódy pre nastavenie premennej a pre získanie jej hodnoty [13].

Na stránke vývojárov je možné vidieť plánované zlepšenia, ktoré by mali priniesť väčší komfort pri používaní nástroja *JSE*. Ich cieľom sú zlepšenia ako napríklad kompatibilita s najnovším *ANT* spúšťačom a *Java* gramatikou, odstránenie nepoužívaného kódu, zlepšenie generovania dokumentácie, uistenie sa či má kód konzistentnú konvenciu, implementácia hygieny a mnoho ďalších [13].

```
public syntax assert {
    case #{
        assert (?expression1:expression);
    }:
    return #{
        do {
            if (!(examples.AssertionsStatus.ASSERTIONS_DISABLED
            || (?expression1)))
            throw new examples.AssertionError();
        } while(false);
    };
    case #{
        assert (?expression1:expression, ?expression2:expression);
    }:
    return #{
        do {
            if (!(examples.AssertionsStatus.ASSERTIONS_DISABLED
            || (?expression1)))
                throw new examples.AssertionError(?expression2);
        } while(false);
    };
}
```

Obrázok 1 – 3 *JSE* párovanie vzorov

```
public syntax property {
    case #{
        ?modifiers:* property ?:type ?:name ?init:*;
    }:
    { Fragment getterName =
            new IdentifierFragment(
            ‘‘get’’.concat(capitalize(name.getName())));
        Fragment setter = #{ };
        Fragment setterName =
            new IdentifierFragment(
            ‘‘set’’.concat(capitalize(name.getName())));
        return #{
            private ?type ?name ?init;
            ?modifiers ?type ?getterName() { return ?name; }
            ?modifiers void ?setterName (?type x) { this.?name = x; }
        };
    }
    private static String capitalize(String s) {
    if (s.length() == 0) {
            return s;
    }
    char chars[] = s.toCharArray();
    chars[0] = Character.toUpperCase(chars[0]);
    return new String(chars);
    }
}
```

Obrázok 1 – 4 *JSE* generovanie zapuzdrenia [13]

1.2 Anotácie a projekt Lombok

Anotácie poskytujú údaje o programe, ktoré nie sú časťou samotného programu. Nemajú žiadny priamy efekt na operácie v kóde, ktorý komentujú. Anotácie môžu byť použité rôznymi spôsobmi. Môžu slúžiť ako informácie pre kompilátor, napríklad pri odhaľovaní chýb alebo potlačení upozornení. Anotácie môžu využívať aj nejaké softvérové nástroje (napr. *Annotation Processing Tool*) na generovanie kódu, *XML* súborov, atď. Platnosť anotácií môže byť buď iba v čase kompilácie, alebo v čase pred zavedením kódu do pamäte, alebo aj počas vykonávania programu. Anotácie smú byť aplikované na programové deklarácie tried, polí, metód a ďalších programových elementov. Anotácie sa smú písať pred programovým elementom na ktorý môžu byť použité a pred menom anotácie musí byť uvedený znak `@`. Niektoré anotácie smú obsahovať aj parametre uvedené v oblých zátvorkách, ktoré sú oddelené čiarkou. Príklad anotácie s parametrami je uvedený na obrázku 1 – 5 [15] [18].

```
@Autor(
   meno = "Viktor Dobransky",
   dátum = "5.1.2012"
)
public class MojaTrieda() { }
```

Obrázok 1 – 5 Anotácia s parametrami [15]

Vytvorenie nového anotačného typu je podobné vytvoreniu rozhrania. Naznačenie, že sa jedná o definovanie nového anotačného typu zabezpečuje kľúčové slovo `@interface`. Ďalej je možné určiť typ parametrov a názov parametrov. Každý názov parametra musí byť ukončený oblými zátvorkami. Je možné tiež určiť prednastavenú hodnotu parametra, ktorá bude použitá v prípade, ak sa explicitne nedefinuje nová hodnota. Prednastavenú hodnotu indikuje kľúčové slovo `default`. Pri vytváraní nového anotačného typu je možné nastaviť aj zotrvanie anotácie, kde sú na výber tri možnosti zotrvania a to v zdrojovom kóde , *class* kóde a počas behu aplikácie. Toto zotrvanie sa nastavuje anotáciou `@Retention`, kde ako parameter sa uvedie jedna

z troch možností zotrvania. Ďalej je možné určiť kde v rámci zdrojového kódu bude možné používať nový anotačný typ. Tu sú na výber niekoľko možností ako napríklad použitie anotácie na typ, pole, metódu, lokálnu premennú, atď. To sa nastavuje anotáciou `@Target`,kde sa ako parameter uvádza pole prvkov s možnosťami uplatnenia anotácie. Príklad vytvorenia nového anotačného typu je uvedený na obrázku 1 – 6 [15] [18].

```
@Retention(RetentionPolicy.CLASS)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Autor {
String autor() default "Viktor Dobransky";
String dátum();
}
```

Obrázok 1 – 6 Nový anotačný typ [15]

Pokročilejšie využitie anotácií je možné vytvorením anotačného procesora, ktorý môže čítať *Java* program a vytvárať akcie založené na anotáciách. Nástroj spracujúci anotácie *APT* (z angl. *Annotation Processing Tool*) je štandardne súčasťou *Java* kompilátora od verzie 6. *APT* hľadá a vykonáva anotačné procesory, ktoré môžu produkovať nový zdrojový kód a ďalšie súbory. Anotačné procesory používajú množinu reflexných *API* (z angl. *Application Programming Interface*) a podporujú infraštruktúru na vykonávanie ich spracovateľných programových anotácií. Reflexné *API* poskytuje v čase kompilácie čitateľné zobrazenie programovej štruktúry. Reflexia je bežne používaná programami, ktoré požadujú schopnosť skúmať, alebo modifikovať správanie aplikácie počas vykonávania a je považovaná za silnú techniku, ktorá dovoľuje aplikáciám vykonávať operácie, ktoré by inak neboli možné [10] [27].

1.2.1 Projekt Lombok

Projekt *Lombok* predstavuje nástroj umožňujúci vďaka anotáciám uľahčiť prácu programátora a sprehľadniť zdrojový kód. Uvádza knižnicu s otvoreným zdrojovým

kódom, umožňujúcu generovanie kódu. *Lombok* nie je len syntaktickým obohatením jazyka, ale predstavuje aj unikátny prístup ku generovaniu kódu a otvára nové vývojárske možnosti. Umožňuje transformáciu abstraktného syntaktického stromu (z angl. *Abstract Syntax Tree - AST*) modifikovaním štruktúry počas kompilácie. *AST* je stromová reprezentácia analyzovaného zdrojového kódu, vytvoreného kompilátorom. Kód generovaný nástrojom *Lombok* je viditeľný pre triedy v rámci tej istej kompilačnej jednotky, na rozdiel od metód rozšírenia, ktoré používajú priamu modifikáciu bajt-kódu. Podporuje tiež viac ako jeden mechanizmus pre vykonávanie generácie kódu, zahrňujúc *Java* anotácie. Použitie *Java* anotácií umožňuje vývojárom modifikovať anotované triedy [6] [19].

Projekt *Lombok* je k dispozícii ako jeden `jar` súbor na stránke projektu. Tento súbor predstavuje inštalátor pre integráciu s *IDE*, ktorý zahrňuje aplikačné programové rozhranie. Na väčšine systémov sa jednoduchým dvojklikom na tento súbor spustí inštalátor. Ak systém nie je konfigurovaný pre správne spustenie `jar` súborov, tak môže byť tento súbor spustený z príkazového riadku. Nástroj je podporovaný iba prostrediami *Eclipse* a *NetBeans*. Názov `jar` súboru musí byť pridaný do súboru `classpath` pre každý projekt v ktorom sa budú používať anotácie projektu [14].

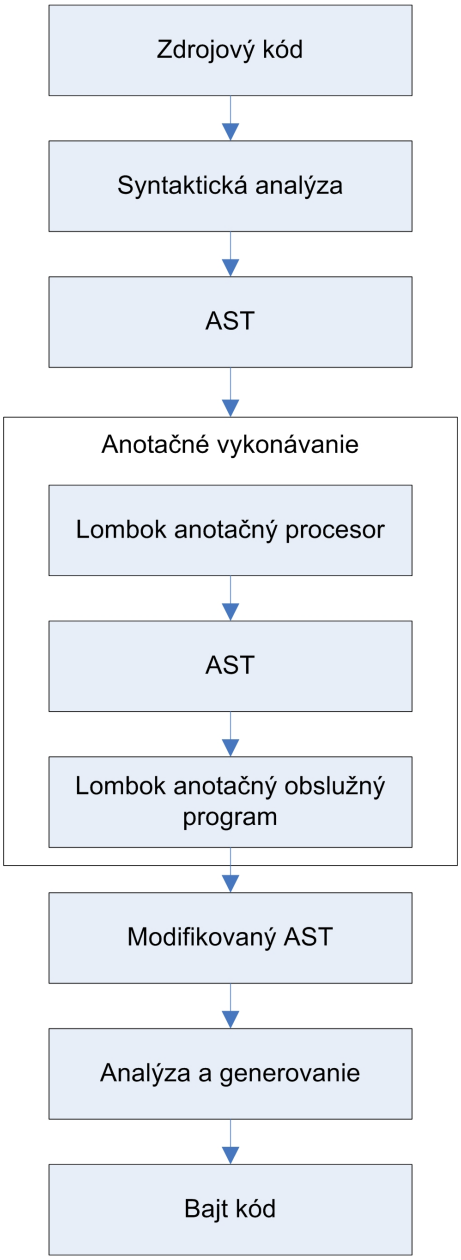
Poskytované anotácie `@Getter` a `@Setter` generujú metódy umožňujúce zapuzdrenie položky. V prípade, ak je položka s názvom `pondelok` a typom `Boolean` , tak po okomentovaní tejto položky anotáciou `@Getter` bude k dispozícií metóda s názvom `isPondelok` vracajúca pravdivostnú hodnotu premennej `pondelok`. V prípade, ak typ tejto položky nie je `Boolean`, tak sa vytvorí metóda s názvom `getPondelok` vracajúca hodnotu premennej `pondelok`. Obdobne je to pri použití anotácie `@Setter`. Pri týchto anotáciách je možné do zátvorky ako parameter uviesť aj prístupovú úroveň novo vytvorenej metódy, pričom pri neuvedení tejto hodnoty je prednastavená hodnota na verejnú viditeľnosť [14]. *Lombok* obsahuje ešte niekoľko anotácií uľahčujúcich prácu programátorov.

Projekt *Lombok* sa spúšťa ako anotačný procesor. Anotačný procesor pôsobí ako

odosielateľ, ktorý vysiela ku anotačným obslužným programom *Lombok-u*. Obslužný program umožňuje spracovať anotačnú požiadavku. Anotačné obslužné programy *Loombok-u* deklarujú špecifickú anotáciu, ktorú spracovávajú. Pri prijímaní údajov obslužným programom, *Lombok* anotačný procesor poskytuje objekt reprezentujúci *AST* anotovaného uzla. Anotované môžu byť triedy, metódy, polia, atď. Anotačný obslužný program *Lombok-u* umožňuje voľne modifikovať *AST* vložením nových uzlov reprezentujúcich metódy, polia a výrazy. Po etape vykonávania anotácií, kompilátor generuje bajt kód z modifikovaného *AST*. Na obrázku 1 – 7 je možné vidieť jednotlivé kroky kompilátora a umiestnenie *Lombok* anotačného procesora a anotačného obslužného programu *Lombok-u* v procese prekladu [29].

Anotačná trieda, obslužný program *Eclipse* a obslužný program *Javac* sú tri základné triedy, ktoré je potrebné vytvoriť pri rozširovaní jazyka pomocou *Lombok* projektu. Pri tvorbe rozšírenia si je potrebné uvedomiť, kde sa použije kľúčové slovo indikujúce rozšírenie a kde sa bude generovať nový zdrojový kód. Obmedzenie rozširovania jazyka pomocou tohto nástroja spočíva v písaní znaku @ pred každým kľúčovým slovom. Avšak je možné napísať tento znak iba raz a anotačným procesorom, ktorý by prechádzal jednotlivé uzly *AST* by sa mohlo hľadať novo definované rozšírenie na ktoré aplikuje generačné pravidlá. To by umožnilo písať rozšírenia iba za použitia jedného znaku indikujúceho spracovanie anotáciami [29].

Trieda obslužného programu je zodpovedná za vytvorenie *AST*, ktorý bude reprezentovať rozšírenie v čistej *Jave* a je zodpovedná za vloženie tohto *AST* do existujúceho *AST* triedy. Keďže *NetBeans* používa *Javac*, tak je potrebné vytvoriť okrem obslužného programu pre *Eclipse* aj *Javac* obslužný program pre *NetBeans*. *Javac* a *Eclipse* obslužné programy musia byť umiestnené v balíku *lombok*. *Lombok* anotačný procesor ich následne môže objaviť. Pre *Eclipse* obslužný program je potrebné pridať anotáciu
`@ProviderFor(EclipseAnnotationHandler.class)` a implementovať rozhranie `EclipseAnnotationHandler`. Obdobne *Javac* obslužný program musí byť anotovaný



Obrázok 1 – 7 Proces prekladu [29]

`@ProviderFor(JavacAnnotationHandler.class)` a musí implementovať `JavacAnnotationHander` rozhranie. Následne je potrebné pre každý obslužný program napísať logiku spracovania, ktorá pozostáva z označení anotácie, že je spracovateľná (iba v prípade *Javac*), ďalej nasleduje vytvorenie zdrojového kódu a jeho vloženie do *AST* anotovanej triedy. Príklad *Javac* obslužného programu triedy umož-

ňujúcej vytvorenie jednoduchej metódy `HelloWorld`, ktorá vypíše do konzoly slová `Hello World` je uvedený na obrázkoch 1 – 8 a 1 – 9 [29].

```
@ProviderFor(JavacAnnotationHandler.class)
@SuppressWarnings("restriction")
public class HandleHelloWorld implements
JavacAnnotationHandler<HelloWorld>{
@Override public boolean handle(
AnnotationValues<HelloWorld> annotation, JCAnnotation ast,
JavacNode annotationNode) {
  JavacHandlerUtil.markAnnotationAsProcessed(annotationNode,
  HelloWorld.class);
  JavacNode typeNode = annotationNode.up();
  JCMethodDecl helloWorldMethod = createHelloWorld(typeNode);
  JavacHandlerUtil.injectMethod(typeNode, helloWorldMethod);
  return true;
 }
```

Obrázok 1 – 8 *Javac* obslúžny program - časť s metódou *handle* [29]

Ďalším krokom je vytvorenie funkcie, ktorá zabezpečí vytvorenie novej metódy. Vkladanie metódy do *AST* je vykonávané *Lombok* pomocnými triedami `JavacHandlerUtil` a `EclipseHandlerUtil`. Pri vytváraní nového uzla *AST* je potrebné uviesť všetky hodnoty potrebné na vytvorenie tohto uzla. Sú to hodnoty ako napríklad návratový typ, parametre, prístupová úroveň, klauzula `throw`, telo metódy a pod. Tiež je možné vkladať ďalšie uzly a tým vytvárať nový *AST*. Jednotlivé metódy pre *Javac* a pre *Eclipse* obslužné programy sa od seba dosť líšia a ich písanie je pomerne obtiažne. Pri tvorbe *Eclipse* obslužného programu sa nepoužíva zapuzdrenie a namiesto reťazcov sa musí používať pole znakov [29].

```
private JCMethodDecl createHelloWorld(JavacNode type) {
 TreeMaker treeMaker = type.getTreeMaker();
 JCModifiers modifiers = treeMaker.Modifiers(Modifier.PUBLIC);
 List<JCTypeParameter> methodGenericTypes =
   List.<JCTypeParameter>nil();
 JCExpression methodType = treeMaker.TypeIdent(TypeTags.VOID);
 Name methodName = type.toName("helloWorld");
 List<JCVariableDecl>  methodParameters   =
   List.<JCVariableDecl>nil();
 List<JCExpression> methodThrows = List.<JCExpression>nil();
 JCExpression printlnMethod = JavacHandlerUtil.chainDots(
   treeMaker, type, "System", "out", "println");
 List<JCExpression> printlnArgs =
 List.<JCExpression>of(treeMaker.Literal("hello world"));
 JCMethodInvocation printlnInvocation = treeMaker.Apply(
   List.<JCExpression>nil(), printlnMethod, printlnArgs);
 JCBlock methodBody = treeMaker.Block(0,
 List.<JCStatement>of(treeMaker.Exec(printlnInvocation)));
 JCExpression defaultValue = null;
 return treeMaker.MethodDef( modifiers,methodName,
   methodType,methodGenericTypes, methodParameters,
   methodThrows, methodBody, defaultValue );
 }
}
```

Obrázok 1 – 9 *Javac* obslúžny program - časť s metódou *createHelloWorld* [29]

1.3 SugarJ

Cieľom *SugarJ* je umožniť programátorom vyjadrovať sa v syntaxi, ktorá je viac zameraná na doménu riešeného problému. Využíva pritom integráciu syntaktického rozšírenia do hlavného rozširovacieho mechanizmu programovacích jazykov - knižníc [26].

SugarJ je druh *Javy*, ktorá podporuje obohatené knižnice (z angl. sugar libraries). Tieto knižnice umožňujú syntakticky rozšíriť knižnice programovacieho jazyka. Rozšírené knižnice vytvárajú syntaktický prínos, ktorý poskytuje používateľom definovanú syntax. Každý kúsok syntaktického obohatenia definuje nejakú rozšírenú syntax a transformáciu rozšírenej syntaxe do syntakticky známeho hostujúceho jazyka. Rozšírené knižnice prinášajú rovnakú flexibilitu ako konvenčné knižnice. Môžu byť použité kdekoľvek importovaním syntaktického obohatenia. Syntax viacerých doménovo špecifických jazykov môže byť zložená importovaním všetkých odpovedajúcich rozšírených knižníc. Obohatené knižnice sú taktiež aplikovateľné samy na seba, to znamená, že každá takáto knižnica môže importovať ďalšie takéto knižnice [9].

Obohatené knižnice riešia jazykové rozšírenia na všetkých metaúrovniach. Metaúroveň rozlišuje definíciu jazyka podľa jeho použitia. To znamená, že jazyková definícia je na vyššej metaúrovni ako program napísaný v tomto jazyku. V tomto zmysle, obohatené knižnice definujúce jazykové rozšírenie sú na vyššej metaúrovni ako programy, ktoré používajú obohatené knižnice [9].

SugarJ je založený na troch existujúcich jazykoch a to na *Jave*, ktorá je použitá ako hosťovský jazyk pre aplikačný kód, na formalizme pre definíciu syntaxe *SDF*, ktorý je použitý na opísanie konkrétnej syntaxe, a na transformačnom jazyku *Stratego*, ktorý je použitý na obohatenie rozšíreného kódu do *SugarJ* kódu. Rozšírený kód nemusí byť preložený iba do *Javy*, ale tiež do *SDF* alebo *Stratego* fragmentov definujúcich ďalšie rozšírenie. Takéto opakované použitie existujúcej technológie dovoľuje implementovať plne vybavený a vysoko expresívny prototyp *SugarJ* [9].

SDF je skratka pre *Syntax Definition Formalism* a slúži na formálny popis syntaxe. *SDF* je modulárny, môže byť použitý na definíciu lexikálnych a bezkontextových gramatík. Je podporovaný generátorom syntaktickej analýzy. Jeho dôraznosť dovoľuje definovanie syntaxe stručne a prirodzene. Modularita *SDF* umožňuje opakované použitie kódu [28].

Stratego je jazyk a sada nástrojov pre transformáciu programov. Jazyk *Stratego* poskytuje pravidlá pre vyjadrenie základných transformácií, programovateľné stratégie prepisovania pre kontrolovanie aplikačných pravidiel, konkrétnu syntax pre vyjadrenie syntaktických vzorov v syntaxi objektového jazyka a dynamicky prepisovateľné pravidlá pre vyjadrenie citlivých kontextových transformácií. Poskytuje teda podporu pre vývoj transformačných komponentov na vysokej úrovni abstrakcie [28].

Pre použitie obohatených knižníc je potrebné iba importovať knižnicu za pomoci definovaného importovacieho príkazu. V súbore, kde je importovaná obohatená knižnica smie programátor používať syntax poskytovanú knižnicou kdekoľvek po importovacom príkaze. Všetky syntaktické konštrukcie z knižnice sú preložené do čistého *Java* kódu automaticky počas kompilácie. Pri písaní obohatených knižníc je potrebné definovať ako bude rozšírený jazyk a ako sa rozšírenie obohatí. Obohatené knižnice sa skladajú z dvoch častí. Prvou je rozšírenie gramatiky hosťujúceho jazyka s novými syntaktickými pravidlami a druhou časťou je preklad nových jazykových konštrukcií do originálneho jazyka. V *SugarJ* programátori definujú obe časti pomocou vysokoúrovňových deklarácií v tvare `public sugar Name { ... }`. Tento tvar obsahuje *SDF* a *Stratego* kód organizovaný do sekcií. Všetky platné *SDF* a *Stratego* sekcie môžu byť použité v obohatenej deklarácií. V sekcií *SDF* nazývanej `context-free syntax` vývojár knižnice môže rozšíriť gramatiku hosťujúceho jazyka s novými syntaktickými pravidlami. Príklad je uvedený na obrázku 1 – 10 [9]. Syntaktické pravidlo špecifikuje pridaný neterminál (definovaný napravo od šípky), vzor pre novo vytvorenú konkrétnu syntax (naľavo do šípky) a meno pre uzol syntaktického stromu vytvoreného týmto produkčným pravidlom. Analogicky

```
context-free syntax
"nil" -> JavaLiteral {cons("Foo"), prefer}
FooTypeId -> JavaTypeName {cons("TypeName"), prefer}
FooTypeName -> FooTypeId {cons("MyId")}
```

Obrázok 1 – 10 *SugarJ* - definovanie syntaktických pravidiel [9]

v sekcií *Stratego* pravidiel vývojár knižnice môže definovať programové transformácie nazývané prekladacie pravidlá (z angl. desugarings). Príklad takéhoto pravidla je možné vidieť na obrázku 1 – 11 [9]. Prekladacie pravidlá sú použité za kľúčovým slo-

```
desugarings
\ Foo -> Null \
\ MyId(s) -> Id("String") \
```

Obrázok 1 – 11 *SugarJ* - prekladacie pravidlá [9]

vom `desugarings`. Takéto pravidlo obsahuje zodpovedajúci vzor (naľavo od šípky) a generačnú šablónu (napravo od šípky). Prekladacie pravidlo označuje programovú transformáciu z rozšíreného do pôvodného jazyka s možnými ďalšími rozšíreniami. Prekladacie pravidlá sú špecifické použitím konkrétnej syntaxe, takže programátor nepotrebuje čítať alebo písať abstraktný syntaktický strom. V poslednej sekcií prekladania obohatenej knižnice, knižničný vývojár deklaruje vstupný bod pre preklad. Po syntaktickej analýze prekladač *SugarJ* aplikuje tieto prekladacie pravidlá spôsobom zdola-nahor, začínajúc listami syntaktického stromu a postupovaním k jeho koreňu. Kompilácia zlyhá ak vstupný program nemôže byť jednoznačne syntakticky analyzovaný za použitia všetkých syntaktických pravidiel, alebo ak nejaké spustené pravidlá signalizujú chybu, alebo ak prekladací program stále obsahuje fragmenty používateľovho rozšírenia [9].

Obohatené knižnice môžu byť skladané importovaním viacerých obohatených knižníc do jedného súboru. *SugarJ* zabezpečuje skladanie gramatických rozšírení a preklad deklarácie obohatených knižníc. Zlúčiteľnosť základných gramatických formalizmov a transformácia jazyka boli hlavnými kritériami, ktoré rozhodovali o postavení

SugarJ na *SDF* a *Stratego*. Skladanie dvoch obohatených knižníc nie je vždy možné bez konfliktov alebo nejednoznačností. To sa môže stať v prípade, ak sa prekrývajú syntaktické rozšírenia [9].

V *SugarJ* importy a deklarácie obohatených knižníc sa môžu objaviť iba na najvyššej úrovni súborov a teda nemôžu byť vnorené vnútri ďalších deklarácií. Preto gramatické zameranie a prekladacie pravidlá sú vždy zarovno s najvyššou úrovňou štruktúry súboru. Každý vysokoúrovňový vstup je syntakticky analyzovaný použitím aktuálnej gramatiky, ktorá odráža všetky obohatené knižnice v aktuálnom rozsahu. Prvým vysokoúrovňovým vstupom aktuálnej gramatiky je inicializačná *SugarJ* gramatika, ktorá zahŕňa syntaktické definície *Javy*, *SDF* a *Stratego*. Výsledok syntaktickej analýzy je heterogénny abstraktný syntaktický strom, ktorý môže obsahovať preddefinované *SugarJ* uzly a uzly definované používateľom. Kompilátor prekladá používateľom definované uzly rozšírení na každom vysokoúrovňovom vstupe do preddefinovaných *SugarJ* uzlov použitím štandardného prekladu. Štandardný preklad pozostáva z prekladacích pravidiel v aktuálnom rozsahu a prekladacích pravidiel z predtým deklarovaných alebo importovaných syntakticky obohatených knižníc. Preklad predstavuje transformácie abstraktného syntaktického stromu, ktoré kompilátor aplikuje metódou zdola-nahor na všetky uzly abstraktného syntaktického stromu dokiaľ nie je dosiahnutý pevný bod. Výsledkom tohto prekladacieho kroku je homogénny abstraktný syntaktický strom, ktorý obsahuje iba uzly deklarované v počiatočnej *SugarJ* gramatike. Nasleduje rozdelenie každej vysokoúrovňovej *SugarJ* deklarácie do *Java* fragmentov, *SDF* a *Stratego* a opakované použitie ich príslušných implementácií[9].

Inštalácia tohto nástroja je pomerne jednoduchá a to aj vďaka návodu uvedeného na stránke projektu. Prvým krokom je nainštalovanie programu za pomoci funkcie *inštalácia nového softvéru*, ktorú obsahuje prostredie *Eclipse*. Potom stačí pridať stránku z ktorej sa má stiahnuť nový softvér a pokračovať intuitívne v inštalácií. Následne stačí pri vytvorení nového projektu iba upraviť súbor `.project` v ktorom

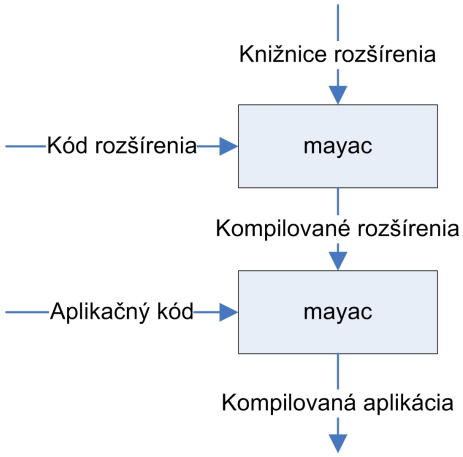
je potrebné zmeniť kompilátor na *SugarJ* kompilátor. Teraz je už možné vytvárať obohatené knižnice. Na stránke projektu sú tiež nejaké príklady možných rozšírení, ktoré je možné stiahnuť, vyskúšať a upravovať [9].

1.4 Maya

Maya je verzia *Javy*, ktorá umožňuje programátorom písať ich vlastné syntaktické rozšírenia, volané *Mayovia* (z angl. Mayans). *Mayovia* môžu reinterpretovať alebo rozšíriť *Mayanskú* syntax expandovaním do ďalšej *Mayanskej* syntaxi. Operujú na abstraktných syntaktických stromoch (z angl. Abstrakt Syntax Tree - *AST*) a ich expanzia je vykonávaná počas syntaktickej analýzy ako sémantické akcie. Vyjadrovacia schopnosť nástroja *Maya* pochádza z upravovania gramatických produkcií na generické funkcie a *Mayov* ako multimetódy na týchto generických funkciách. *Mayovia* sú definovaný použitím bohatej množiny parametrov s danou špecializáciou. *Mayovia* môžu byť odoslaný na typ uzla *AST*. Môžu tiež predstavovať statický typ výrazu, tokenovú hodnotu alebo subštruktúru akéhokoľvek *AST* uzla. Viacnásobné odosielanie dovoľuje používateľom rozširovať sémantiku jazyka prepisovaním základných akcií jazyka [3].

Makro rozšírenia, ktoré autori nazvali *Mayovia*, sú písané ako kód, ktorý generuje AST. Keď sú *Mayovské* rozšírenia skompilované môžu byť hneď načítané *mayac* kompilátorom a použité v ďalšom kompilovaní aplikačného kódu 1 – 12. *Mayské* rozšírenia sú odoslané zo syntaktického analyzátora aplikácie v čase kompilácie, kde môžu reinterpretovať alebo rozšíriť *Maya* syntax expandovaním sa do ďalšej *Maya* syntaxe. Syntaktické rozšírenia môžu byť použité na vloženie doménovo špecifického jazyka do existujúceho jazyka [3].

Mayské rozšírenia pôsobia na abstraktnú syntax. Nástroj *Maya* sa môže uistiť, že tieto rozšírenia produkujú správny *AST* a to vďaka tomu, že uzly tohto stromu sú typové. *Maya* dovoľuje programátorom generovať *AST* so šablónami. Tie môžu byť použité na zostavenie ľubovoľných kúskov abstraktnej syntaxe. *Mayské* rozšírenia majú lexikálne určené oblasti platností mien. Ich lokálne deklarácie môžu zaznamenať stav obvodových inštancií a ich deklarácie sú oddelené od importov. Importované *Mayské* rozšírenia sú použiteľné len v rozsahu ich importov. Tieto funkcie



Obrázok 1 – 12 Kompilácia *Maya* prekladačom [3]

dovoľujú veľkú flexibilitu vďaka tomu, že syntaktické transformátory zdieľajú stav a sú vystavené základnému kódu. Vďaka týmto funkciám *Maya* vytvára tri nové implementačné techniky [3].

Prvou z nich je podpora odosielania statických typov. *Maya* podporuje lenivú (z angl. lazy) typovú kontrolu s lenivou syntaktickou analýzou. Typy a abstraktné syntaktické stromy sú vypočítané na požiadanie. Lenivá typová kontrola dovoľuje *Mayským* rozšíreniam odoslanie statických typových argumentov a vytvorenie variabilných väzieb, ktoré sú viditeľné pre ďalšie argumenty. Posledné argumenty musia byť typovo otestované a až potom môžu byť zviazané. Lenivá syntaktická analýza dovoľuje *Mayským* rozšíreniam aby boli importované do ľubovoľného bodu programu. Syntax, ktorá nasleduje po importácií tohto rozšírenia musí byť analyzovaná lenivým spôsobom po *Mayských* definíciách rozšírenia pre všetky nové produkcie. Lenivé vyhodnotenie syntaktických stromov je vystavené explicitne programátorovi *Mayského* rozšírenia, takže efekt lenivosti môže byť kontrolovaný [3].

Druhou implementačnou technikou je nová technika syntaktickej analýzy, ktorá sa volá vzorová syntaktická analýza a to podľa staticky testovaných tiel šablón pre syntaktickú korektnosť. Technika vzorovej syntaktickej analýzy dovoľuje programátorovi použiť kvázi úvodzovky (z angl. quasiquotes) na generovanie každého správneho

AST [3].

Nástroj *Maya* podporuje hygienu a referenčnú transparentnosť v šablónach prostredníctvom premenovania krokov počas kompilácie. Takéto premenovanie je možné vďaka tomu, že zviazané konštrukcie musia byť explicitne deklarované v *Maya* gramatike.

Maya môže byť použitá na implementáciu makier slúžiacich na jazykové rozšírenia ako sú *MultiJava* a aspekty. Poskytuje makro knižnicu, ktorá obsahuje funkcie ako formátovanie tlačených reťazcov, kompresia syntaxe pre stavbu polí a kolekcií, a syntax cyklu `foreach`, ktorý ich prechádza [3].

Syntaktické rozšírenie je definované dvomi časťami. Prvou je nová *LALR* produkcia, ktorú je potrebné pridať v prípade, ak nová syntax nie je akceptovaná existujúcou gramatikou. Druhou je definovanie *Mayských* rozšírení sémantických akcií pre produkciu. Jednoduchý príklad zápisu deklarácie produkcie je na obrázku 1 – 14. Kľúčové slovo `abstract` indikuje definovanie produkcie. Kľúčové slovo `syntax` indikuje definovanie gramatickej produkcie a teda *Mayského* rozšírenia. `Statement` je návratový typ produkcie, ktorý reprezentuje príkaz alebo skupinu príkazov. Argumenty produkcie sú pravou stranou produkčného pravidla. Všeobecnejší zápis pravidla je možné vidieť na obrázku 1 – 13. Kľúčové slovo `lazy` znamená, že sa jedná o lenivú syntaktickú analýzu. Väčšina symbolov v *Maya* gramatike sú typové uzly *AST*. Produkcie a *Mayské* rozšírenia smú byť definované len nad takýmito uzlovými symbolmi, avšak je tu aj podpora niekoľkých druhov parametrických gramatických symbolov, ktoré sú použité na definovanie opakovania a kontroly lenivej syntaktickej analýzy [3].

```
Statement -> MenoMetódy( Formality ) lazy-blok
```

Obrázok 1 – 13 Produkčné pravidlo [3]

Vďaka tomuto rozšíreniu je možné písať cyklus `foreach` spôsobom uvedeným na obrázku 1 – 15. Cyklus `foreach` je možné použiť na kľúčové hodnoty objektu typu

```
abstract Statement
EForEach(Expression:Enumeration eExp
  \.Foreach(Formal var)
  lazy(BraceTree, BlockStmts) body)
{
Final StrictTypeName castType =
StrictTypeName.make(var.getType());
return new Statement {
  for(Enumeration eVar=$eExpr;eVar.hasMoreElements(); ){
    $(DeclStmt.make(var))
    $(Reference.makeExpr(var.getLocation()))=
    ($castType) eVar.nextElement();
    $body
  }
};
}
```

Obrázok 1 – 14 Implementácia rozšírenia [3]

Hashtable.

```
h.keys().foreach(String ex){
  System.out.println(ex+"="+h.get(ex));
}
```

Obrázok 1 – 15 Ukážka použitia rozšírenia cyklu [3]

Tento zápis sa pri preklade expanduje do podoby uvedenej na obrázku 1 – 16, pričom identifikátor `eVar` sa nezobrazuje v zdrojovom kóde programu.

```
for(Enumeration eVar$ = h.keys();eVar$.hasMoreElements(); ){
  String ex;
  ex = (String) eVar$.nextElement();
  System.out.println(ex+"="+h.get(ex));
}
```

Obrázok 1 – 16 Expandovanie príkazu [3]

Mayské rozšírenia môžu byť použité pre široké spektrum aplikácií, od jednoduchých makier v jazyku *C* až po jazykové rozšírenia všeobecného použitia. *Maya* je pravdepodobne najužitočnejšia pre úlohy niekde medzi týmito extrémnymi zložitosťami [3].

1.5 AHEAD Tool Suite

AHEAD Tool Suite (ďalej len ATS) je komplexná množina nástrojov pre tvorbu jazykov. Všetky tieto nástroje sú písané v rozšírenej verzie *Javy* volanej tiež *Jakarta*, alebo skrátene iba *Jak*. Sada nástrojov Jakarta je množinou kompilačných nástrojov pre rozširovanie priemyselných programovacích jazykov doménovo špecifickými konštrukciami [4].

AHEAD (*Algebraic Hierarchical Equations for Application Design*) je model, ktorý zovšeobecňuje porovnávanie špecifikácií viacerých programov a viacerých nekódových reprezentácií. *AHEAD* je ukážkou toho, že softvér môže mať elegantnú, hierarchickú matematickú štruktúru, ktorá je vyjadrená vnorenými množinami rovníc [4].

ATS je množina *Java* nástrojov, ktoré podporujú vývoj programov v *AHEAD*. *ATS* je používaný na konštrukciu simulátorov požiarnej podpory v americkej armáde a v samotnej syntéze *ATS*. Kľúčovou vlastnosťou *AHEAD* je škálovateľnosť. Neexistujú žiadne limitácie na veľkosť programov, alebo počet reprezentácií, ktoré môžu byť syntetizované. Množina programov, ktorú *ATS* môže zostaviť zahŕňa *XML* dokumenty, programy napísané v rozšírenej *Jave*, stavové automaty, jazykové gramatiky, návrhové pravidlá a výrazy [5].

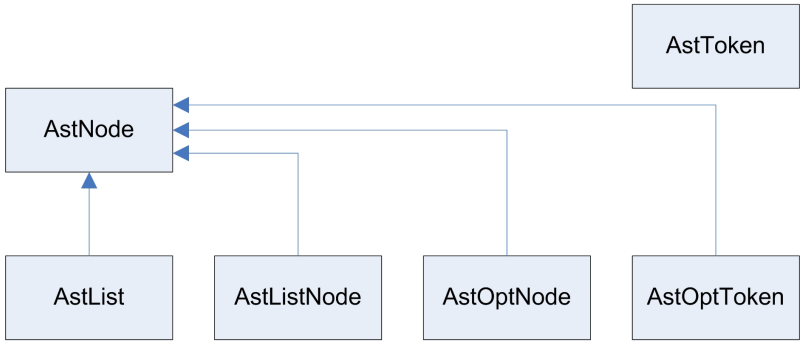
Existujú dva spôsoby v ktorých *AHEAD* môže byť použitý na tvorbu jazyka a to buď skonštruovať priamo prekladač, alebo vyvinúť preprocesor, ktorý preloží program jazyka do iného hostiteľského jazyka, ktorým môže byť napríklad jazyk *Java*. Obe techniky sú si podobné [1].

Pri rozširovaní jazyka *Java* preprocesorom je prvým krokom vytvoriť *Ant XML* súbor pre tvorbu nového jazyka. *AHEAD* poskytuje dva *XML* súbory, a to `Template.xml` a `Configure.xml`. Vďaka nim je možné generovať ďalšie *Ant* súbory. `Template.xml` predstavuje *XML* šablónu pre budovanie jazyka. `Configure.xml` zoberie súbor `Template.xml` ako vstup a podľa neho generuje vlastný *Ant XML* súbor,

ktorý je následne použitý pri kompilačnom procese. V prípade, ak sa rozširuje hosťovský jazyk, tak model tohto jazyka už existuje. Ak je teda model jazyka známy, je ďalej potrebné vytvoriť určitú adresárovú štruktúru pre novú jazykovú špecifikáciu a sémantické akcie. Nasledujúcim krokom je definovanie gramatického rozšírenia vkladaného do hosťovského jazyka. Na jeho definovanie sa používa *Bali LL(k)* gramatická špecifikácia jazyka. Tá je navrhnutá na produkciu syntaktických analyzátorov pre *ATS*. Tento súbor je potrebné vložiť do daného priečinka. Ďalšou akciou, ktorú je potrebné zabezpečiť, je vytvorenie súboru rovností. *AHEAD* syntetizuje program vyhodnocovaním rovností, ktoré sa skladajú z gramatických a sémantických vrstiev jazyka. Je teda potrebné určiť gramatické vrstvy jazyka *Java* a nového rozšírenia, a k ním prislúchajúce sémantické akcie. Pri vytváraní novej sémantickej vrstvy je možné použiť nástroj *bali2layer*, ktorý slúži na vytvorenie akýchsi šablón sémantických súborov s príponou `jak`, pričom ako vstupný argument sa použije *Bali* gramatika nového rozšírenia. Tieto šablóny je možné po vygenerovaní upraviť tak, ako to vývojár vyžaduje. Funkcia týchto súborov spočíva v získaní argumentov abstraktného syntaktického stromu (z angl. *Abstract Syntax Trees - AST*) z uzla *AST* a umožnením tlačenia výstupu a teda samotného prekladu. Posledným krokom je samotné vytvorenie preprocesora. To zabezpečí nástroj *Ant*, kde ako parameter sa uvedie vlastný *Ant XML* súbor, ktorý bol vygenerovaný v prvých krokoch. Takto vytvoreným preprocesorom je možné transformovať súbory *Java* s vlastným rozšírením do čistých *Java* súborov už bez rozšírenia [1].

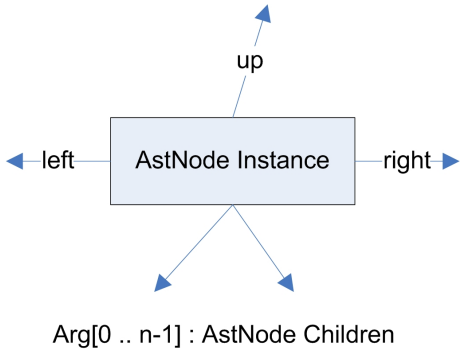
AHEAD ponúka programové schopnosti pre vytváranie a manipuláciu s *AST Javy*. Opiera sa o hierarchiu rozličných druhov *AST* uzlov. Podstatu tejto hierarchie je možné vidieť na obrázku 1 – 17. *AstNode* je koreň *AST* hierarchie tried. Všetky tokeny sú inštanciou triedy *AstToken*. Všetky generované triedy z *Bali* gramatiky sú podtriedy triedy *AstNode* [1].

AstNode má pole pozostávajúce z nula alebo viacerých potomkov. Naľavo a napravo sú prepojené so súrodencami a uzol vyššie je rodič. Ukazovateľ na rodiča má hod-



Obrázok 1 – 17 Typy *AST* uzlov [1]

notu `null` v prípade, ak *AST* uzol nemá rodiča. Podobne je to so súrodencami. Terminálny uzol *AST* nemá žiadnych potomkov. Naznačenie je na obrázku 1 – 18. Existuje 22 metód, ktoré môžu byť použité nad uzlami *AST*. Patria sem metódy ako napríklad metóda na konvertovanie *AST* uzla na reťazec znakov, metóda na klonovanie uzla *AST*, metóda na nájdenie rodičovského uzla podľa mena, nahradenie, zmazanie, úpravu alebo odpojenie *AST* uzla, atď [1].



Obrázok 1 – 18 Inštancia uzla *AST* [1]

AST triedy sú generované z *Bali* gramatických špecifikácií. Uvažujme nasledujúce *Bali* produkčné pravidlo [1].

```
Neterminál1 : TOKEN1 Neterminál2 TOKEN2 TOKEN3 Neterminál3 :: ATrieda ;
```

Bali generuje triedu s menom *ATrieda*. Inštancia triedy *ATrieda* je koreňom, ktorý má dvoch potomkov a to inštanciu triedy *Neterminál2* a inštanciu triedy *Neterminál3*. Oba tieto objekty sú voči sebe v súrodeneckom vzťahu a sú uložené v poli

s názvom `arg`. Analogicky tokeny sú uložené v poli s názvom `tok`. Obsahy poľa `arg` a poľa `tok` pre daný príklad sú uvedené nižšie [1].

```
arg[0] = // ukazovateľ na inštanciu Neterminál2
arg[1] = // ukazovateľ na inštanciu Neterminál3
tok[0] = // ukazovateľ na inštanciu TOKEN1
tok[1] = // ukazovateľ na inštanciu TOKEN2
tok[2] = // ukazovateľ na inštanciou TOKEN3
```

Jadro metaprogramovania pozostáva zo schopnosti konštruovať a skladať kódové fragmenty. Na to slúžia kódové konštruktory (z angl. code constructor) a kódové úniky (z angl. code escapes). Kódové konštruktory konvertujú špecifikovaný kódový fragment do *AST*. Hodnota konštruktora je ukazovateľom na koreň *AST*. Napríklad výrazový konštruktor `exp{...}exp` uzatvára syntakticky správny *Java* výraz. Keď je konštruktor vykonaný, je vytvorený *AST* pre výraz a koreňom tohto stromu je výsledok. Analogicky konštruktor `stm{...}stm` predstavuje vytvorenie príkazového *AST*. Príklad je uvedený na obrázku 1 – 19. Aktuálne existujú 17 rôznych kódových konštruktorov [1].

```
Exp a = exp{ 10 + b*8 }exp;
AST_Stm c = stm{ Metoda(); if (d < 15) return e; }stm;
System.out.println(a); // výsledok "10 + b*8"
System.out.println(c); // výsledok "Metoda(); if(d <15) return e;"
```

Obrázok 1 – 19 Príklad kódových konštruktorov [1]

Únikové kódy umožňujú skladanie *AST*. Únikové kódy nahradzujú premennú v danom mieste kódového konštruktora. Označenie únikového kódu začína znakom $ a pokračuje príslušnou odpovedajúcou skratkou. Príklad použitia únikového kódu vkladajúceho výraz prislúchajúci premennej `a` do *AST* príkazu je uvedený na obrázku 1 – 20 [1].

Objekt, ktorý je použitý na prechádzanie *AST* daného uzla v aktuálnom čase je nazývaný *AST* kurzor. Tieto kurzory sú inštanciou triedy *AstCursor*. Sú využívané

```
Exp a = exp{ 10 + b*8}exp;
AST_Stm c = stm{ d = ($exp(a)) * 13;}stm;
System.out.println(c); // výsledok "d = (10 + b*8) *13;"
```

Obrázok 1 – 20 Príklad kódového úniku [1]

na hľadanie príslušného *AST* a po jeho označení je možné tento *AST* nahradiť, vymazať, upraviť alebo dokonca odpojiť. Príklad uvedený na obrázku 1 – 21 ilustruje prechádzanie jednotlivých uzlov *AST*, pričom po nájdení uzla s názvom *PrimExpr* ho nahradí za uzol definovaný v premennej `e` [1].

```
AstCursor c = new AstCursor();
for (c.FirstElement(e); c.MoreElement(); c.NextElement() ) {
  if (c.node instanceof PrimExpr) {
    copy = (AST_Exp) e.clone();
    c.Replace(copy);
  }
}
```

Obrázok 1 – 21 Použitie *AST* kurzora [1]

Existuje desať metód, ktoré je možné aplikovať na inštanciu kurzora. Patria sem metódy na nastavenie kurzora na danú pozíciu, posun kurzora, zistenie, či sa kurzor môže ďalej posunúť, posun kurzora o pozíciu vyššie v *AST*, mazanie, nahradenie *AST* na ktorý aktuálne ukazuje kurzor, atď [1].

1.5.1 Bali

Bali je *LL(k)* gramatická špecifikácia jazyka navrhnutá na produkciu syntaktických analyzátorov pre sadu nástrojov *AHEAD*. Na rozdiel od tradičných gramatických špecifikácií, *Bali* podporuje kompozíciu špecifikácií, takže pod-gramatiky môžu byť zdieľané a znovu použité. Táto gramatická špecifikácia kombinuje upravené *LL(k)* *BNF* (*Backus - Naur Form*) gramatické pravidlá s lexikálnymi definíciami a vložený *Java* kód. Prvé pravidlo definujúce *Bali* gramatiku v *BNF* je uvedené na obrázku 1 – 22 [22].

```
BaliGrammar : [Options] [ParserCode] [Statements] :: BaliGrammarNode ;
```

Obrázok 1 – 22 Definovaná *Bali* gramatika [22]

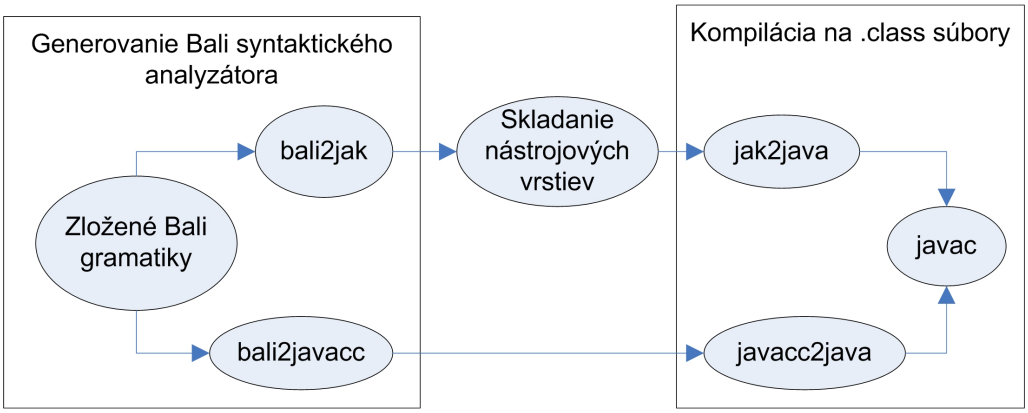
Označenie návestia `BaliGrammar` predstavuje neterminálny symbol, ktorý je nasledovaný dvojbodkou a tiež predstavuje štartovací symbol pre gramatiku. Neterminálne symboly musia byť identifikátormi. Ďalej nasleduje množina pozostávajúca z jednej, alebo viacerých gramatických produkcií oddelených vertikálnou čiarou a ukončených bodkočiarkou. Ukážka 1 – 22 má iba jednu gramatickú produkciu. Konečná fráza tejto produkcie je za dvomi dvojbodkami, v tomto prípade `BaliGrammarNode`. Táto fráza predstavuje meno produkcie. Toto meno je ďalej použité ako meno triedy pri párovaní syntaktických elementov špecifikovaných v produkcií počas syntaktickej analýzy. Produkcia samotná je sekvencia troch symbolov a to `Options`, `ParserCode` a `Statements`. Každý z týchto symbolov je voliteľný, čo je označené hranatými zátvorkami obklopujúcimi každý symbol. Toto pravidlo definuje hranice *Bali* špecifikácie gramatiky. *Bali* gramatika začína sekciou `options`, nasledovanou sekciou `parser code` a končiacou sekciou `statements`. Podstata *Bali* gramatiky je v sekcií `statements`, ktorá je definovaná v jazyku *Bali* tak, ako to naznačuje obrázok 1 – 23 [22].

Tento príklad ukazuje ďalšie dva *BNF* pravidlá a to `Statements` a `Statement`. Prvé pravidlo má iba jednu produkciu a v tomto prípade definuje jednoduchý zoznam

```
Statements  : (Statement) +;
Statement   : RequireStatement
            | BaliGrammarRule
            | BaliTokenDefinition
            | JavacodeProduction
            | RegexTokenDefinition
            | TokenManagerDeclarations
            ;
```

Obrázok 1 – 23 Definovaná Bali gramatika [22]

elementov `Statement`. Jednoduchý zoznam je vždy naznačený oblými zátvorkami obklopujúcimi symbol a následným znamienkom +. Pravidlo Statement pozostáva zo šiestich produkcií z ktorých každá má presne jeden neterminálny symbol. Keďže tieto pravidlá sú bez mena, môžu byť nazývané pod produkcie a predstavujú akýsi podtyp produkčného pravidla. Význam týchto dvoch pravidiel je, že ak existujú príkazy v *Bali* gramatike, tak musia obsahovať jeden alebo viac príkazov, pričom môžu byť buď požadovaným príkazom, *Bali* gramatickým pravidlom, *Bali* definíciou tokenu, kódovou produkciou *Java*, regulárnym výrazom na definíciu tokenov alebo hlavná tokenová deklarácia. *Bali* syntaktický analyzátor je generovaný z jednej, alebo viacerých *Bali* gramatických špecifikácií a výsledná syntaktická analýza je obyčajne integrovaná do nástroja *AHEAD* v podobe prekladača. Konštrukcia takéhoto nástroja pozostáva z troch fáz, ktoré sú znázornené na obrázku 3 – 19. Prvou fázou je použitie nástrojov *balicomposer*, *bali2jak* a *bali2javacc* na generovanie zdrojového kódu pre *Bali* syntaktický analyzátor. Výstupom tejto fázy sú rozobraná stromová vrstva v zdrojovom kóde *Jak* a *JavaCC* syntaktický analyzátor. Druhá fáza pozostáva z kompozície vygenerovanej rozobratej stromovej vrstvy s ďalšími vrstvami nástroja *AHEAD*. Treťou fázou je preklad generovaného zdrojového kódu do podoby *Java* `class` súborov [22].



Obrázok 1 – 24 ATS generovanie [22]

1.6 Provonanie nástrojov

Nástroje *Lombok*, *Maya*, *ATS* a *JSE* využívajú pri rozširovaní syntaxe jazyka *Java* *AST*. Výhoda takéhoto prístupu spočíva v lepšej organizácií vkladaných údajov, resp. čítaných údajov. Tieto údaje sú v tomto prípade celé časti *AST*. Manipulácia s takýmito stromami je jednoduchá. Nevýhodou takéhoto prístupu je snáď iba to, že je potrebné poznať jazykovú špecifikáciu daného programovacieho jazyka. Takáto definícia môže byť do značnej miery rozsiahla. Na druhej strane nie je vždy potrebné využívať celé spektrum tejto definície, ale iba určité potrebné konštrukcie.

Nástroj *Lombok* nebol pôvodne určený na rozširovanie syntaxe *Javy*, ale iba na uľahčenie písania zdrojového kódu, napríklad generovaním zapuzdrenia a pod. Možnosti tohto nástroja sú v porovnaní s nástrojmi priamo určenými na prácu s jazykom o niečo menšie. *SugarJ* a *Lombok* sú primárne určené pre programovacie prostredie *Eclipse* a teda je to ich mínusovým faktorom.

Maya a *ATS* sú určené nie len pre rozširovanie jazyka *Java*, ale aj pre rozširovanie iných jazykov, ako napríklad *C*, *SQL*, *XML* a pod. Poskytujú široké možnosti pre prácu s *AST* daného jazyka. Vďaka týmto širokým možnostiam je možno tieto nástroje považovať za výhodnejšie pri rozširovaní jazyka *Java* zložitými konštrukciami. Avšak takáto výhoda je daň za väčšiu zložitosť pri ich používaní. *JSE*, *SugarJ*

a *Lombok* sú určené len pre *Javu* a žiaden iný programovací jazyk. Zo spomínaných nástrojov iba *JSE* a *SugarJ* boli vytvorené priamo s úmyslom rozširovania syntaxe *Javy*.

Z hľadiska ich inštalácie sú všetky tieto nástroje pomerne ľahko inštalovateľné. Väčšinou je potrebné iba prekopírovať potrebné súbory, prípadne upraviť nejaké súbory ručne alebo nainštalovať daný program za pomoci *Eclipse marketplace*. Riešenie hygieny v *JSE*, *ATS* a *Maya* je ďalším plusovým faktorom, ktorý zaváži pri výbere nástroja na tvorbu rozšírenia.

V praktickej časti práce bude na tvorbu rozšírenia použitý nástroj *ATS*. Tento nástroj bol vybratý vďaka tomu, že pracuje s jednoduchou *Bali* gramatikou, využíva stromovú štruktúru s ktorou sa ľahko pracuje, obsahuje definíciu syntaxe jazyka *Java*, ktorá je potrebná k ďalšiemu rozširovaniu jazyka, na stránke projektu sú kvalitné ukážky jednoduchých rozšírení jazyka *Java* a je schopný generovať sémantické šablóny. Neposlednými kladnými vlastnosťami nástroja sú implementovaná hygiena a intuitívne vytváranie nových *AST*.

2 Integrovanie dopytov do programovacieho jazyka

2.1 Microsoft LINQ

LINQ (z angl. *Language INtegrated Query*) je metodológia, ktorá uľahčuje a zjednocuje implementáciu akéhokoľvek druhu prístupu k dátam. *LINQ* je programovací model, ktorý predstavuje dopyty ako prvotriedny koncept akéhokoľvek *Microsoft .NET* jazyka. Pre kompletnú podporu je potrebné v jazyku doplniť rozšírenia. Tieto rozšírenia zvýšia produktivitu, poskytnutím kratšej, významnejšej a výraznejšej syntaxe pri manipulácií s dátami. *LINQ* je možné aplikovať na kolekcie objektov, *XML*, *SQL*, kolekciu dát a entity. Pri jeho použití na entitný dátový model vzniká abstraktná vrstva, ktorá je nezávislá od fyzickej dátovej vrstvy [17].

Asi najlepšou možnosťou ako pochopiť použitie nástroja *LINQ* je ukážka na príkladoch. Každý z uvedených príkladov predpokladá, že premenná `zakaznici` je zoznam objektov. Za kľúčovým slovom `var` sú premenné `poziadavka` a `expr`. Prekladač ho využíva ako informáciu o tom, že musí sám určiť typ premennej. *SQL* požiadavky sa v príkaze píšu za znamienkom = [17].

Príklad 2 – 1 ilustruje prechádzanie všetkých objektov zoznamu `zakaznici`, pričom pri každej iterácií prechádzania sa uloží aktuálne počítaná hodnota do premennej `c`. Typ tejto premennej nie je potrebné definovať, keďže program si ho vie sám odvodiť. Identifikátor tejto premennej sa píše za kľúčovým slovom `from`. To, ktorý zoznam sa bude prechádzať musí byť za kľúčovým slovom `in` a byť identifikátorom zoznamu. Vyberané údaje je možné za pomoci kľúčového slova `where` obmedzovať na základe podmienky, resp. viacerých podmienok uvedených za týmto kľúčovým slovom. Tieto podmienky majú byť syntakticky obdobné ako pri použití klasického príkazu `if`. Je ich možné spájať za použitia spojovníkov `&&` a `||` a prípadne zoskupovať zátvorkami. Každý objekt sa testuje na základe podmienky a pri jej splnení je možné hodnotu

ďalej použiť. Poslednou klauzulou v tomto príklade je klauzula `select`. Tá označuje, ktorá premenná, resp. skupiny premenných, majú byť vybraté do konečného zoznamu. Tento konkrétny príkaz vyberá všetkých zákazníkov, ktorí sú zo Slovenska [17].

```
var poziadavka =  from c in zakaznici
                  where c.Krajina =="Slovensko"
                  select c;
```

Obrázok 2 – 1 Ukážka *LINQ* príkazu

Ukážka 2 – 2 sa líši od predchádzajúcej v tom, že výsledný zoznam údajov bude zoradený podľa mena zákazníka. Zoradenie výsledného zoznamu je naznačené za kľúčovým slovom `orderby`. Tu by mal byť uvedený typ, ktorý je možné porovnávať napríklad *String*, *Integer* a pod. Pri zoraďovaní je možné uviesť viacero kritérií, ktoré sú oddelené čiarkou, pričom ak sa nepodarí určiť, ktorý z údajov bude „vyššie“, resp. „nižšie“ vo výslednom zozname, tak sa použije ďalšia z podmienok zoradenia. Zoradenie údajov je v abecednom poradí, no v prípade, ak je požadované opačné zoradenie údajov je možné použiť kľúčové slovo `descending` uvedené za vymenovanými zoraďovacími kritériami. Ďalším faktorom, v ktorom sa líši tento príklad od predchádzajúceho je, že za klauzulou `select` je za pomoci kľúčového slova `new` vybratých viacero položiek zo zoznamu a teda nie celý objekt typu `zákazník`. Takto je možné vyberať položky, s ktorými sa plánuje ďalej pracovať. V tomto prípade je to iba meno a mesto z ktorého zákazník pochádza. Nakoniec tento zoznam ešte bude zoradený podľa mien zákazníkov v abecednom poradí [17].

```
var poziadavka =  from c in zakaznici
                  where c.Krajina =="Slovensko"
                  orderby c.Meno
                  select new { c.Meno, c.Mesto};
```

Obrázok 2 – 2 Ukážka *LINQ* príkazu

Ďalší príklad 2 – 3 poukazuje na to, že je možné opakovať určité časti príkazu.

V tomto prípade sa jedná o opakovanie klauzuly `from` a teda možnosť presnejšej špecifikácie. Príkaz vyberá všetky objednávky pre produkty s identifikačným číslom tri [17].

```
var poziadavka =  from p in produkty
                  where p.IDprodukt == 3
                  from o in p.Objednavky
                  select o;
```

Obrázok 2 – 3 Ukážka *LINQ* príkazu

Nasledujúci príkaz 2 – 4 zoskupuje zákazníkov podľa krajiny pôvodu. Za klauzulou `group` sa nachádza to, čo sa bude zoskupovať a za klauzulou `by` to, podľa čoho sa budú zoskupovať údaje. V tomto prípade premenná `expr` obsahuje zoznam párových položiek. Prvou položkou je kľúč, podľa ktorého sa zgrupovali údaje a druhá položka sú údaje prislúchajúce tomuto kľúču. Nasleduje aj ilustrácia ako možno prechádzať jednotlivé položky skupiny. Klasickým prechádzaním všetkých položiek príkazom `foreach` je možné robiť iteráciu cez každú položku a získať buď kľúčovú hodnotu položky za použitia klauzuly `Key`, alebo celú hodnotu skupiny pre odpovedajúci kľúč. Táto celá hodnota má charakter zoznamu [17].

```
var expr =  from c in zakaznici
            group c by c.Krajina;

foreach(IGrouping<Krajiny, Zakaznik> zakaznikSkupina in expr) {
  Console.WriteLine("Kraijna: {0}, zakaznikSkupina.Key);
  foreach(var polozka in zakaznikSkupina) {
    Console.WriteLine(polozka);
  }
}
```

Obrázok 2 – 4 Ukážka *LINQ* príkazu

Táto diplomová práca vychádza z týchto jednoduchých príkazov. Existujú ešte nejaké klauzuly, ktoré *LINQ* používa na prácu s údajmi, avšak pre potreby diplomovej

práce budú postačovať iba tie uvedené v ukážkach. Mnohé z neuvedených príkazov je možné implementovať aj bez rozšírenia syntaxe jazyka.

2.2 Existujúce riešenia podobné rozšíreniu *LINQ* pre jazyk *Java*

V súčasnosti existujú rôzne doplnkové riešenia pre jazyk *Java*, ktoré umožňujú prácu s databázou, objektmi a prípadne *XML* súbormi formou podobnou nástroju *LINQ* od spoločnosti *Microsoft*. Niektoré podporujú všetky spomínané možnosti, iné zase iba jednu z možností. Jednou z hlavných myšlienok týchto nástrojov je to, aby programátor nemusel písať dopyt formou reťazca, v ktorom by vznikol priestor pre neúmyselné chyby. Takisto pomoc nástroja pri písaní dopytu (automatické dopĺňanie zdrojového kódu) je možné považovať za výhodu. Vlastnosť písať aplikačnú logiku priamo v *Java* zdrojovom kóde vedie k jednoduchšej a prehľadnejšej práci s údajmi. Každý z týchto nástrojov sa snaží priblížiť myšlienke nástroja *LINQ* od *Microsoftu*, avšak ich syntax je dosť odlišná. Asi najväčšiu podobu v syntaxi s rozšírením *LINQ* majú nástroje *Iciql* a *Querydsl*, ale pri ich používaní je stále potrebné písať aj iné drobnosti okrem samotného príkazu na selekciu údajov. Teda všetky tieto možnosti písania dopytov sa iba približujú čistému *SQL* dopytu. Preto cieľom tejto práce je vytvoriť rozšírenie, ktoré bude mať čo najväčšiu podobu syntaxe s nástrojom *LINQ* a jazykom SQL.

2.2.1 JaQue

Je jedným z nástrojov, resp. knižnicou umožňujúcou poskytovanie *LINQ* schopností na *Java* platforme. Použitím *ASM* tvorí výrazové stromy, ktoré sú použité na zostavenie dopytu v doménovo špecifickej technológií alebo jazyku. *ASM* je viacúčelový nástroj na manipuláciu a analýzu *Java* bajtkódu. Aktuálne je podporované písanie dopytu nad objektmi a *XML* súbormi. Autori projektu pracujú aj na podpore *JPA*

(*Java Persistence API*). Príklad časti zdrojového kódu je možné vidieť na obrázku 2 – 5. Takéto selektovanie údajov je trocha neštandardné a podobnosť s jazykom *SQL* je minimálna [2] [12].

```
for (String s : select(nums, { int t => Integer.toString(t) }))
        System.out.print(s);
for (int i : where(nums, { int t => t < 2 }))
        System.out.print(i);
```

Obrázok 2 – 5 Ukážka JaQue použitia[12]

2.2.2 Iciql

Poskytuje databázový prístupový obal pre *JDBC* (*Java DataBase Connectivity*). *JDBC* je *Java API*, ktoré dovoľuje *Java* programom vykonávať *SQL* príkazy. Veľkosť nástroja je pomerne malá (125KB) a bez žiadnych spúšťacích závislostí. Nástroj zabezpečuje typovú bezpečnosť v čase kompilácie. Za plus je možné považovať automatické dopĺňanie zdrojového kódu v *IDE*. Aktuálne je tento nástroj pod rýchlym vývojom a preto je možné, že sa zmení *API* a konfigurácia v každom jeho novom vydaní. Aktuálne podporuje databázy *H2*, *HSQLDB*, *Derby*, *MySQL* a *PostgreSQL*. Príklad časti zdrojového kódu je možné vidieť na obrázku 2 – 6. Tento prístup k databázovým údajov sa už viac podobá jazyku *SQL* [11].

```
Product p = new Product();
List<Product> restock =
db.from(p).where(p.unitsInStock).is(0).select();
```

Obrázok 2 – 6 Ukážka Iciql použitia [11]

2.2.3 QueryDSL

Tento nástroj dovoľuje konštrukciu typovo bezpečných *SQL* dopytov nad *JPA*, *JDO* (*Java Data Object* - *Java* dátové objekty) a *SQL* v *Jave*. Namiesto písania dopytov

ako reťazca, alebo extrahovaním ich do *XML* súborov, sú dopyty konštruované cez elegantné *API*. Podpora doplňovania kódu je pri tomto nástroji samozrejmosťou. Nedovoľuje písať takmer žiadne syntakticky nesprávne dopyty, pretože je tu typová kontrola na všetkých úrovniach. Má lepšiu adaptáciu na refaktorizačné zmeny v doménových typoch a ľahšie inkrementálne definovanie dopytov. Tento nástroj bol testovaný s použitím nástroja *Hibernate* a databázami *Derby*, *HSQLDB* a *MySQL*. V prípade *JDO* bol nástroj testovaný s *DataNucleus* a *AppEngineDatastore* od spoločnosti *Google*. Oba projekty poskytujú produkty pre manažment aplikačných dát v prostredí *Java*. V prípade použitia *SQL* dopytov je podpora databáz *Derby*, *H2*, *HSQLDB*, *MySQL*, *Postgres*, *Oracle* a *MS SQL Server*. Taktiež je možné používať dopyty na selekciu údajov z *Java* kolekcií. Príklad časti zdrojového kódu 2 – 7 reprezentuje selekciu údajov pri použití *SQL* dialektu. Obdobné použitie je aj pri práci s *JDO*, *JPA* a *Java* kolekciami. Tento nástroj má pravdepodobne najrozsiahlejšie použitie oproti ostatným spomínaným nástrojom [21] [7] [8].

```
QCustomer customer = new QCustomer("c");
SQLTemplates dialect = new HSQLDBTemplates();
SQLQuery query = new SQLQueryImpl(connection, dialect);
List<String> lastNames =
    query.from(customer)
    .where(customer.firstName.eq("Jan"))
    .list(customer.lastName);
```

Obrázok 2 – 7 Ukážka QueryDSL zdrojového kódu [21]

2.2.4 SBQL4J

SBQL4J (*Stack-Based Query Language for Java*) je ďalším rozšírením pre jazyk *Java* založený na *SBA* (*Stack-Based Architecture*). *SBA* je formálna metodológia adresovania objektovo orientovaných databázových dopytov a programovacích jazykov. *SBQL4J* umožňuje spracovávať *Java* objekty za pomoci dopytov. Tento nástroj sa snaží zachovávať softvérové princípy ako sú modularita, minimálnosť, univerzálnosť,

typová bezpečnosť a čisté, precízne sémantiky. Zdrojový kód s dopytmi je syntakticky analyzovaný preprocesorom integrovaným s *OpenJDK* kompilátorom. Je tesne integrovaný s *Javou*, čo znamená, že *Java* premenné môžu byť použité v dopytoch priamo, dopyty vracajú *Java* objekty, *Java* metódy a konštruktory môžu byť zavolané vnútri dopytov. Jazyk dopytov je typovo bezpečný, pretože dopyty sú testované v čase kompilácie. Dopyty môžu byť preložené do čistého *Java* kódu, takže vykonávanie môže byť v konečnom dôsledku veľmi rýchle. Príklad časti zdrojového kódu je uvededný na obrázku 2 – 8 [25] [23].

```
List<Customer> customers = getCustomerList();
List<String> customerOrders = #{
  (customers as c).
  ($index as custIndex, c.orders as o).
  ("Zakaznik "+(custIndex + 1)+" ma poradove cislo "+o.orderID)
};
```

Obrázok 2 – 8 Ukážka SBQL4J použitia [23]

3 Rozšírenie jazyka Java o dopyty

Prvou možnosťou ako postupovať pri tvorbe rozšírenia je využitie vodopádového modelu vývoja softvéru. Tento model predstavuje sekvenčný proces, pri ktorom jednotlivé fázy vývoja na seba nadväzujú. V tomto prípade to znamená vytvorenie čo najpresnejšej a čo najsprávnejšej definície syntaxi nového rozšírenia. Z tejto syntaxe je potrebné určiť preklad, resp. sémantický význam pre každú novú položku rozšírenia. Nasleduje programovacia fáza a po nej fáza testovacia. Takýto postup je vhodný v prípade ak už existujú určité skúsenosti s programovaním rozšírenia pre jazyk *Java* za pomoci nástroja *ATS* a je teda možné predvídať chyby už vo fáze plánovania a definovania syntaxe a sémantiky. Slabinou tohto riešenia je to, že ak sa vyskytne chyba, ktorá bude zanedbaná v nejakej z počiatočných fáz, tak táto chyba sa prenesie až do etapy testovania. V prípade ak táto chyba vplýva na ostatné časti programu, bude potrebné opraviť široké spektrum programových závislostí, čo môže byť časovo náročné.

Druhým spôsobom ako postupovať pri tvorbe rozšírenia je využitie inkrementálneho prístupu vo vývoji softvéru. V tomto prípade sa jedná o postup pri ktorom sa budú vytvárať jednotlivé funkcie rozšírenia jazyka *Java* postupne. Navrhne sa syntax a sémantika pre jednu jednoduchú časť, implementuje sa, otestuje sa a chyby sa hneď opravia. Postupne sa tento proces opakuje s novými časťami až kým aplikácia nemá finálnu podobu. Dôležité pri tomto postupe je vedieť zoradiť jednotlivé navrhované časti podľa závislostí. To znamená, že ak funkčná časť A je závislá od funkčnej časti B, tak časť A je možné vyvíjať a implementovať až vtedy, keď je úplne hotová časť B. Ak sa to nepodarí takto rozdeliť, tak môže nastať problém s tým, že bude potrebné upravovať aj diely, ktoré boli predtým skontrolované a funkčné. Výhodou tohto prístupu je, že sa chyby odhaľujú v každej inkrementálnej etape a následne sa môžu rýchlo opraviť a teda sa neprenášajú až do koncových fáz vývoja rozšírenia. Ďalšou preferenciou je, že tento postup je výhodný pri prvej práci s ešte neprebádaným nástrojom *ATS* a gramatickou špecifikáciou *Bali*.

Reálne bol pri tvorbe rozšírenia použitý druhý inkrementálny spôsob, avšak v tejto práci bude prezentovaná prvá metóda a to kvôli jej čitateľnejšej a logickejšej štruktúre.

3.1 Návrh syntaxe rozšírenia jazyka

Pri definovaní syntaxe nového rozšírenia bol ako inšpirácia použitý nástroj *LINQ* od spoločnosti *Microsoft*. Cieľom je zachovanie čo najväčšej syntaktickej podoby s týmto nástrojom avšak nebude kladený striktný dôraz na dodržanie všetkých jeho vlastností. Rozdiel bude napríklad v písaní kľúčových klauzúl. Oproti nástroju *LINQ*, kde sa píšu kľúčové slová iba malými písmenami, tu sa budú písať iba veľkými písmenami. Vďaka takémuto obmedzeniu bude možno dosiahnuť omnoho lepšiu čitateľnosť zdrojového kódu, avšak na úkor jeho zložitejšieho písania.

Na definovanie syntaxe je použitý jazyk *Bali*, ktorý je opísaný v sekcii 1.5.1. Po zistení, že dodaná definícia syntaxe jazyka *Java* (`/dsl/java/JavaGram/grammar.b`) neobsahuje syntaktickú podporu generických typov bolo potrebné túto podporu doplniť. Syntax nového rozšírenia bude napísaná v súbore `grammar.b`, ktorý je potrebné vytvoriť v adresári `/dsl/java/JaiqGram/`.

Na definíciu novej syntaxe je potrebné poznať niektoré produkčné pravidlá *Java* gramatiky, ktoré sú uvedené v súbore `/dsl/java/JavaGram/grammar.b`. Tieto pravidlá sú písané intuitívne a teda je možné sa v nich pomerne ľahko orientovať. Pri tejto práci boli použité tieto elementy z *Java* gramatiky: `Expression` (výraz), `Statement` (príkaz), `AST_TypeName` (dátový typ, napr. `String`, `Integer`, `Person`,atď), `AST_Modifiers` (viditeľnosť, napr. `public`, `private`,atď), `AST_VarDecl` (deklarovanie premennej), `AllocExprChoices` (parametre udávané pri alokácií premennej novým dátovým typom), `AllocationExpression` (alokačný výraz uvedený v príkaze za znamienkom =), `VariableDeclaratorId` (identifikátor premennej), `QName` (kvalifikované meno). Pridanie novej gramatiky podporujúcej generické typy je uve-

dené na obrázku 3 – 1. `ColType` predstavuje neterminálny symbol, ktorý spája staré dátové typy a novo vkladané dátové typy. Tomuto spojeniu je možné priradiť ľubovoľné meno, ktoré je uvedené za dvojitými dvojbodkami. V tomto prípade je to `CollecTypes`. Toto meno by v prípade potreby predstavovalo meno súboru umožňujúceho definovanie sémantiky. Každé produkčné pravidlo môže začínať s klauzulou `LOOKAHEAD`. Tá sa používa na rozlíšenie gramatiky v prípade podobne začínajúcich produkčných pravidiel. Symboly uvedené v úvodzovkách sú terminálne symboly. Názvy sémantických tried uvedené za dvojitou dvojbodkou sú v tomto prípade nepodstatné, pretože sa celý vstupný kód iba prepíše na výstup bez akejkoľvek sémantickej zmeny.

```
AST_TypeName : LOOKAHEAD(2) ColType :: CollecTypes ;
ColType      : AST_QualifiedName "<" Params :: ColTypeTwo ;
Params       : AST_QualifiedName [NextParam]">" :: ParamsM ;
NextParam    : "," AST_QualifiedName  :: NextParamM ;
AllocationExpression : LOOKAHEAD("new" ColType())
    "new" ColType AllocExprChoices :: AllocCollect ;
```

Obrázok 3 – 1 Definovanie syntaxe pre podporu generických typov

Po vytvorení produkčných pravidiel pre podporu generických typov je možno definovať nový dátový typ `var` , ktorý je použitý pri selekcií údajov. Pre triedu zabezpečujúcu spracovanie tohto typu je možné zvoliť ľubovoľné meno. V tomto prípade má trieda názov `MojVar` a bude zabezpečovať sémantické rozbalenie príkazu. `VAR` je token, ktorý je definovaný na začiatku súboru spôsobom uvedeným nižšie. Každý token musí byť definovaný na vlastnom riadku.

```
"var" VAR
Statement : LOOKAHEAD(2) [AST_Modifiers] VAR AST_VarDecl ";" :: MojVar ;
```

Nasledujúca časť sa zaoberá priamo syntaxou *SQL* príkazu na selekciu údajov. Tento príkaz sa píše za znamienkom = a reprezentuje tiež alokačný výraz. Celý príkaz na selekciu údajov môže pozostávať z povinných položiek a nepovinných položiek uvedených v hranatých zátvorkách. Minimálny príkaz musí obsahovať klauzuly `FROM`,

`IN` a `SELECT`. Voliteľné sú klauzuly `WHERE`, `GROUP BY INTO`, `ORDERBY` a `DESCENDING`. Tieto kľúčové slová boli definované obdobne ako klauzula `var` (napr.: `"FROM" FROM`).

```
AllocationExpression : LOOKAHEAD(2)
  FromWheres [GroupWhere] [Order] ASelect :: ALEMain ;
```

Nasledujúce produkčné pravidlo hovorí o tom, že sa môže viacnásobne opakovať použitie klauzúl `FROM` a `WHERE` s tým, že minimálne raz musí byť volaný neterminál `FromWhere`.

```
FromWheres : (FromWhere)+ ;
```

Produkčné pravidlo uvedené nižšie uvádza rozdelenie na pravidlá `From` a `Where`, pričom `Where` je voliteľným pravidlom. Je tu možné vidieť záväznosť nasledovania oboch klauzúl.

```
FromWhere : From [Where] :: FromWhereM ;
```

Nasledujúce produkčné pravidlo definuje, že za kľúčovým slovom `FROM` musí byť uvedený kvalifikovaný názov premennej a za ňou bude pokračovať klauzula `IN` vytváraná produkčným pravidlom `AInto`. Kvalifikovaný názov premennej predstavuje dočasný identifikátor, ktorý bude použitý pri iteratívnom prechádzaní daným zoznamom a teda bude meniť svoju hodnotu pri každom kroku iterácie.

```
From : FROM QName AInto:: FromM ;
```

Produkčné pravidlo uvedené nižšie značí, že za kľúčovým slovom `IN` musí byť uvedený kvalifikovaný názov premennej, ktorý reprezentuje identifikátor zoznamu. Tento zoznam bude použitý ako vstupný zoznam z ktorého sa budú vyberať prvky.

```
AInto : IN QName :: AIntoM ;
```

Nasledujúce produkčné pravidlo určuje čo (`What`) sa má vyberať do nového zoznamu. To čo sa vyberie bude za klauzulou `SELECT`. Možnosti výberu sú definované produkčným pravidlom `What`.

```
ASelect : SELECT What :: SelecM ;
```

To čo je možné označiť ako výber do nového zoznamu, určuje produkčné pravidlo uvedené nižšie. Na výber sú dve možnosti. Prvou je použitie akéhokoľvek výrazu (napr. `person`, `osoba = person`) a druhou je použitie viacerých výrazov oddelených

čiarkou za pomoci nového kľúčového slova `new` a vlnitých zátvoriek.

```
What  : LOOKAHEAD(2) Expression :: WhatMQ
      | LOOKAHEAD("new" "{" ListQName() "}")
        "new" "{" ListQName "}" :: WhatMN ;
```

Pri selektovaní údajov je potrebné rozhodnúť, ktoré údaje budú vyberané na základe nejakého obmedzenia. Na to slúži kľúčové slovo `WHERE` a za ním uvedené podmienky. Jeho definícia je v nasledujúcom produkčnom pravidle.

```
Where : WHERE Expression :: WhereM ;
```

Nasledujúce produkčné pravidlo umožňuje, aby pri zoskupovaní údajov bolo možné definovanie obmedzenia za použitia klauzuly `WHERE`.

```
GroupWhere : Group [Where] :: GroupWhereM ;
```

Produkčné pravidlo uvedené nižšie slúži na definovanie zoskupenia údajov. Pri zoskupovaní dát preto musia byť použité klauzuly `GROUP`, `BY`, `INTO` v takto uvedenom poradí a nasledované príslušnými údajmi. To, čo sa bude zoskupovať je uvedené za klauzulou `GROUP`. Podľa čoho sa budú údaje zoskupovať je uvedené za klauzulou `BY`. Prístup ku skupine združených údajov poskytuje kvalifikovaný názov premennej uvedený za klauzulou `INTO`. Tento prístup k skupine je len v rámci jedného selektívneho príkazu a teda nie je viditeľný mimo tohto príkazu.

```
Group   : GROUP What BY What INTO QName  :: GroupM ;
```

Ďalšie produkčné pravidlo slúži na definíciu zoradenia údajov. Za klauzulou `ORDERBY` musí byť uvedený zoznam kvalifikovaných mien obsahujúci minimálne jeden prvok. Voliteľná klauzula `DESCENDING` slúži na inverzné obrátenie výsledného zoradeného zoznamu dát.

```
Order : ORDERBY ListQName [DESCENDING] :: OrderM ;
```

Produkčné pravidlo uvedené nižšie prezentuje zreťazenie výrazov za pomoci čiarky a teda vytvorenie zoznamu výrazov. Tento zoznam je použitý pri zoraďovacom produkčnom pravidle (`Order`).

```
ListQName : Expression ("," Expression)* ;
```

Keďže dodaná definícia syntaxe *Javy* neobsahuje aj skrátený zápis prechádzania zoznamu za pomoci cyklu `for`, tak to je potrebné doplniť. Pretože sa pridáva nový

príkaz do produkčného pravidla pre príkazy s názvom `Statement`, tak aj toto nové pravidlo bude mať rovnaký názov, a teda názov `Statement`. Z príkladu uvedeného v komentári by malo byť zrejmé, aký druh príkazu sa dopĺňa.

```
// for ( Object o : list)
Statement :
LOOKAHEAD("for" "(" AST_TypeName() VariableDeclaratorId() ":")
"for" "(" AST_TypeName VariableDeclaratorId ":" VariableDeclaratorId ")"
Statement :: ObjectForStm ;
```

Nasledujúce produkčné pravidlo slúži na podporu prechádzania zoznamu vytvoreného za pomoci nového rozšírenia. Myšlienka je obdobná ako pri produkčnom pravidle uvedenom vyššie. V tomto prípade je však typ zoznamu pevne daný (`var`). Dôvod prečo je potrebné vytvoriť aj takéto produkčné pravidlo je ten, že je potrebné priradiť sémantiku novej syntaxi, oproti predchádzajúcemu prípadu, kde sa doplnila iba syntaktická podpora.

```
// for ( var o : list)
  Statement :
  LOOKAHEAD("for" "(" "var" VariableDeclaratorId() ":" )
  "for" "(" VAR VariableDeclaratorId ":" VariableDeclaratorId ")"
  Statement   :: ForVarStm ;
```

3.2 Definovanie rozbalenia

Pri tvorbe rozšírenia jazyka je potrebné určiť zdrojový kód, ktorý bude reprezentáciou daného rozšírenia v čistej *Jave*. Tento kód je potrebné odvodiť od syntaxe a sémantiky pridávaného rozšírenia. Na jeho určenie je možné vytvoriť vzorové príklady a podľa nich vybrať argumenty potrebné na generovanie. Z týchto argumentov sa potom vytvorí sémantický model. Asi najjednoduchším príkazom na selekciu údajov je príkaz uvedený nižšie. Jeho myšlienka spočíva v prechádzaní zoznamu `people` a vytváraní toho istého zoznamu s iným názvom.

```
var b = FROM person IN people SELECT person;
```

V čistej *Jave* by tomuto príkazu odpovedal zdrojový kód uvedený na obrázku 3 – 2. Keďže typ zoznamu (`Person`) nie je možné určiť z daného príkazu, je potrebné vytvoriť metódu, ktorá zabezpečí jeho nájdenie. Tá bude spomenutá v sekcií 4.7. Pretože dočasné premenné je potrebné zachovávať iba v rámci jedného príkazu, tak celý generovaný zdrojový kód okrem výsledného zoznamu je uzavretý do pravdivostnej podmienky (`if(true){ ... }`). Tá zabezpečí potrebnú viditeľnosť premenných.

```
List<Person> b = new ArrayList<Person>();
if(true){
  for (Person  person :  people){
    b.add( person);
  }
}
```

Obrázok 3 – 2 Odpovedajúci zdrojový kód v čistej *Jave*

Ďalšia ukážka 3 – 3 spočíva v selekcií údajov na základe nejakej podmienky uvedenej za klauzulou `WHERE`. Štandardne tu môže byť viacero podmienok avšak je možné sa na nich stále pozerať ako na jeden výraz.

```
var c = FROM person IN people
        WHERE person.getName().startsWith("v")
        SELECT person;
```

Obrázok 3 – 3 Ukážka príkazu na selekciu dát s podmienkou

Prevod tohto príkazu s podmienkou je uvedený na obrázku 3 – 4. Podmienka sa jednoducho prepíše do príkazu `if`. Ako parameter sa uvedie obmedzujúci výraz uvedený za klauzulou `WHERE`. V prípade splnenia podmienky sa pridáva daný prvok do výsledného zoznamu.

Ďalej je potrebné určiť, ako sa bude generovať zdrojový kód pri prechádzaní viacerých zoznamov súčasne. Tento prípad je znázornený na obrázku 3 – 5.

Toto je možné riešiť tak, že sa budú jednotlivé príkazy do seba vnárať, tak ako je to naznačuje 3 – 6. Inou možnosťou je, aby sa najprv prechádzali zoznamy pomocou

```
List<Person> c = new ArrayList<Person>();
if(true){
  for (Person  person :  people){
    if( person.getName().startsWith("v")){
      c.add( person);
    }
  }
}
```

Obrázok 3 – 4 Odpovedajúci zdrojový kód v čistej *Jave* pre 3 – 3

```
var r = FROM a1 IN numbers
        WHERE a1.toString().length() == 1
        FROM b1 IN numbers
        WHERE a1 < b1
        SELECT a1;
```

Obrázok 3 – 5 Zdrojový kód prechádzajúci dva zoznamy súčasne

príkazu `for` a potom by sa aplikovali podmienky. Tie by bolo možné dokonca spojiť do jednej výslednej podmienky. V tomto prípade bol však použitý prvý spôsob a to kvôli tomu aby sa zbytočne neprechádzalo zoznamom, ktorý nebude splňovať prvotnú podmienku.

V prípade použitia klauzúl `GROUP`, `BY` a `INTO` je základných 25 možností ako bude vyzerať výsledný zdrojový kód zoskúpenia. Za klauzulou `INTO` je možné napísať iba kvalifikované meno, avšak za klauzulami `GROUP` a `BY` je možné písať buď výraz (`person`, `person.getName()`, `new ResultPerson(person)`, a pod.), alebo skupinu výrazov
(`new {Meno = person.getName(), Priezvisko = person.getLastName()}`,
`new {person.getName(), person.getLastName()}`, a pod.). Teda za klauzulami `GROUP` a `BY` je výber z piatich základných možností. Príklad príkazu so zoskupením údajov je znázornený na obrázku 3 – 7.

Vygenerovaný kód je možné rozdeliť na dve časti, a to na časť zabezpečujúcu zo-

```
List<Integer> r = new ArrayList<Integer>();
if(true){
  for (Integer  a1 :  numbers){
    if( a1.toString().length() == 1){
      for (Integer  b1 :  numbers){
        if( a1 < b1){
          r.add(a1);
        }
      }
    }
  }
}
```

Obrázok 3 – 6 Odpovedajúci zdrojový kód v čistej *Jave* pre 3 – 5

```
var g = FROM n IN numbers
        GROUP n BY n.toString().startsWith("1") INTO skupina
        SELECT new { a = skupina.key , b = skupina };
```

Obrázok 3 – 7 Selekcia dát zo zoskúpením

skupenie a časť zabezpečujúcu prepis zoskupenia do výsledného zoznamu. Na zoskupovanie údajov je použitá kolekcia `HashMap`, kde ako kľúč je uvedená trieda `SelectContainer`. Táto trieda bude slúžiť na uchovanie výsledných dát, alebo dát potrebných pri zoskupovaní údajov. Bude obsahovať potrebné privátne premenné, ich zapúzdrenie a metódy na porovnávanie inštancií tejto triedy (*hash*, *equals*). Tu je nastavovaná položka `key` danou hodnotou v prípade ak sa jedná o výraz uvedený za klauzulou `BY`. V prípade ak kľúčom je skupina výrazov, tak sú nastavované jednotlivé položky triedy `SelectContainer` osobitne. Hodnoty, ktoré odpovedajú danému kľúču sú uložené v zozname `ArrayList`. Tu sa najprv zistí, či už `HashMap` obsahuje daný kľúč a podľa toho sa rozhoduje ďalej. V prípade k neobsahuje ešte daný kľúč, tak sa vytvorí nový prázdny zoznam. Ak už existuje, tak je potrebné získať hodnoty odpovedajúceho kľúča. Následne sa pridajú dané položky do tohto zoznamu. Potom sa výsledný zoznam hodnôt uloží k odpovedajúcemu kľúču. Druhou časťou je pre-

pis zoskupovaných údajov do výsledného zoznamu. Tu sa jednoducho prechádzajú všetky kľúče daného zoskupeného zoznamu, im odpovedajúce hodnoty a prepisujú sa do výsledného zoznamu. V prípade ak sa v `SELECT` príkaze objaví názov zoskupeného zoznamu spojený bodkou s kľúčovým slovom `key`, tak v prepise sa získava a nastavuje kľúč. V prípade ak sa použije iba názov zoskupovanej skupiny, tak sa získava a nastavuje hodnota odpovedajúceho kľúča. Príklad rozvinutia príkazu 3 – 7 je znázornený na obrázku 3 – 8.

V prípade použitia klauzuly `ORDERBY` je potrebné vytvoriť vlastnú funkciu porovnávania. Na to dobre poslúži trieda `Comparator` a jej metóda `compare`. Táto metóda porovnáva dva vstupné prvky. Príklad uvedený nižšie, ktorý je iba časťou iného príkazu, bude najprv porovnávať mená osôb a v prípade, že nie je možné sa rozhodnúť na základe mien, tak sa bude rozhodovať na základe priezvisk osôb.

```
ORDERBY p.getMeno(), p.getPriezvisko()
```

Pre tento prípad je možné napísať porovnávaciu funkciu, ktorú je možné vidieť na obrázku 3 – 9. V tomto príklade sa porovnávajú prvky triedy `SelectContainer`, pretože výsledný zoznam bude tohto typu. V prípade ak návratová hodnota prvého porovnania je nulová, tak sa použije porovnanie nasledujúceho parametru. V prípade ak aj ten vracia nulovú hodnotu a už neexistuje ďalší porovnávací parameter, tak sa vráti práve táto nulová hodnota.

Následne je potrebné použiť pripravenú zoraďovaciu funkciu na daný zoznam. V prípade ak je použitá klauzula `GROUP`, tak sa zoraďujú hodnoty uvedené pod daným kľúčom. Ak nie je táto klauzula použitá, tak sa zoraďujú údaje výsledného zoznamu. Samotný príkaz zabezpečujúci volanie funkcie zoradenia nad zoznamom je znázornený nižšie.

```
Collections.sort( f, comparator6);
```

V prípade, ak je použité kľúčové slovo `DESCENDING` indikujúce reverzné obrátenie zoradeného zoznamu, tak sa za príkazom zoraďujúcim výsledný zoznam použije príkaz

```
List<SelectContainer1>  g = new ArrayList<SelectContainer1>();
if(true){
  HashMap<SelectContainer1,List<Integer>> skupina8=
  new HashMap<SelectContainer1,List<Integer>>();

  for (Integer  n :  numbers){
    SelectContainer1 selectContainer9 = new SelectContainer1();
    selectContainer9.setkey( n.toString().startsWith("1"));
    List<Integer> listSelectTried10;
    if(!skupina8.containsKey(selectContainer9)){
      listSelectTried10 = new ArrayList<Integer>();
    }else{
      listSelectTried10 = skupina8.get(selectContainer9);
    }
    listSelectTried10.add( n);
    skupina8.put(selectContainer9,listSelectTried10);
  }
  Set<SelectContainer1> keySet11 = skupina8.keySet();
  for (SelectContainer1 key : keySet11){
    SelectContainer1 selectContainer9 = new SelectContainer1();
    selectContainer9.seta( key );
    selectContainer9.setb(skupina8.get(key));
    g.add(selectContainer9);
  }
}
```

Obrázok 3 – 8 Odpovedajúci zdrojový kód v čistej *Jave* pre 3 – 7

na jeho obrátenie uvedený nižšie. Tým sa dosiahne požadované poradie prvkov.

```
Collections.reverse(f);
```

Celkovo môže mať príkaz na selekciu údajov za klauzulou `SELECT` päť základných možností výberu údajov. Podľa týchto údajov je možné určiť typ daného výsledného zoznamu. V prípade, ak je tento typ zistiteľný, tak sa použije zistená hodnota. Ak tento typ nie je zistiteľný, tak sa použije ako dátový typ trieda `SelectContainer`.

```
Comparator<SelectContainer1> comparator6=
new Comparator<SelectContainer1>() {
  @Override
  public int compare(SelectContainer1 o1, SelectContainer1 o2 ){
    Comparable pgetMeno1 = (Comparable) o1.getMeno();
    Comparable pgetMeno2 = (Comparable) o2.getMeno();
    Comparable fgetPriezvisko1 = (Comparable) o1.getPriezvisko();
    Comparable fgetPriezvisko2 = (Comparable) o2.getPriezvisko();

    if(persongetMeno1.compareTo(fgetMeno2) == 0) {
      return pgetPriezvisko1.compareTo(pgetPriezvisko2);
    }
    return pgetMeno1.compareTo(fgetMeno2);
  }
};
```

Obrázok 3 – 9 Odpovedajúci zdrojový kód v čistej *Jave* pre zoradenie údajov

Tento typ sa tiež používa pri selekcií viacerých údajov.

Príklad uvedený nižšie prechádza zoznam s názvom `people`, ktorý je typu `Person`. Myšlienka tohto príkazu je jednoduchá. Kopíruje údaje z jedného zoznam do druhého.

```
var b = FROM person IN people SELECT person;
```

Prepis tohto príkazu v jazyku *Java* je možné vidieť na obrázku 3 – 10. Tu je možné vidieť, že novo vytváraný zoznam je typu `Person`. Tento typ sa dá zistiť na základe porovnania výrazu za klauzulou `SELECT` a výrazu za klauzulou `FROM`. Keďže sa rovnali, tak sa hľadal typ zoznamu `people`. Ten, ak je definovaný v triede alebo v metóde, tak obsahuje aj daný dátový typ. V prípade, ak by sa tento typ nepodarilo zistiť, tak sa použije ako výsledný typ `SelectContainer`.

Ďalší príklad 3 – 11 poukazuje na možnosť definovania výsledného typu alokačným výrazom za pomoci klauzuly `new`. Tento príklad prechádza zoznam s názvom `people`

```
List<Person> b = new ArrayList<Person>();
if(true){
  for (Person  person :  people){
    b.add( person);
  }
}
```

Obrázok 3 – 10 Odpovedajúci zdrojový kód v čistej *Jave*

a vytvára nový zoznam s objektmi typu `ResultPerson`, kde ako argument na konštrukciu objektu je použitý práve prvok zoznamu `people`.

```
var d = FROM person IN people
        SELECT new ResultPerson(person);
```

Obrázok 3 – 11 Selekcia dát spolu s vytváraním nových inštancií

Rozbalenie tohto príkazu do čistej *Javy* je možné vidieť na obrázku 3 – 11. Výraz za klauzulou `SELECT` určuje výsledný typ zoznamu, ktorý je `ResultPerson`.

```
List<ResultPerson> d = new ArrayList<ResultPerson>();
if(true){
  for (Person  person :  people){
    d.add( new ResultPerson(person));
  }
}
```

Obrázok 3 – 12 Odpovedajúci zdrojový kód v čistej *Jave* pre 3 – 11

Ďalší príkaz 3 – 13 vyberá údaje zo zoznam s názvom `people` a vytvára nový zoznam pozostávajúci z mien osôb. Písmená z ktorých pozostávajú tieto mená ešte pred uložením zväčší. V tomto prípade je problém zistiť typ výsledného zoznamu. Tu by bol potrebný prístup k ďalším triedam a teda aj mimo prekladaného súboru.

```
var e = FROM person IN people
        SELECT person.getName().toUpperCase();
```

Obrázok 3 – 13 Ukážka časti zdrojového kódu s nezistiteľným typom

Rozbalenie zdrojového kódu uvedeného na ukážke 3 – 13 je znázornené na obrázku 3 – 14. Keďže výsledný typ nie je možné zistiť, tak sa použije ako výsledný typ `SelectContainer`. Ten obsahuje privátnu premennú s názvom `name` a jej prislúchajúcimi metódami umožňujúcimi zapuzdrenie. Pri prechádzaní zoznamu s názvom `people` sa vytvorí inštancia triedy `SelectContainer` a použije sa metóda `setName` na nastavenie premennej name príslušnou hodnotou. Táto inštancia triedy sa nakoniec uloží do výsledného zoznamu dát.

```
List<SelectContainer1> e = new ArrayList<SelectContainer1>();
if(true){
  for (Person  person :  people){
    SelectContainer1 selectContainer5 = new SelectContainer1();
    selectContainer5.setName( person.getName().toUpperCase());
    e.add(selectContainer5);
  }
}
```

Obrázok 3 – 14 Odpovedajúci zdrojový kód v čistej *Jave* pre 3 – 13

Predposlednou možnosťou, ktorú je možné písať za klauzulou `SELECT` je možnosť výberu viacerých dát s nastavovaním konkrétnych premenných. Príklad uvedený nižšie prechádza zoznam s názvom `people` a selektuje z neho iba mená a priezviska osôb. K týmto položkám je možný ďalší prístup za použitia metód `getMeno` a `getPriezvisko`. Je možné si všimnúť, že názvy týchto metód sú odvodené od názvov použitých pri výbere hodnôt.

```
var f = FROM person IN people
        SELECT new {
         Meno = person.getName(),
         Priezvisko = person.getLastName()};
```

Obrázok 3 – 15 Selekcia viacerých dát

Tomuto zdrojovému kódu odpovedá zdrojový kód uvedený na obrázku 3 – 16. Keďže sa vyberajú viaceré dáta, tak výsledný typ zoznamu je `SelectContainer`. Pri kaž-

dej iterácií zoznamom `people` sa vytvorí inštancia triedy `SelectContainer`, kde sa nastavia potrebné premenné. Názvy metód umožňujúcich nastavenie príslušných položiek sú odvodené od názvov uvedených vo výbere hodnôt. Odpovedajúce hodnoty sú uvedené za znamienkom `=`. Nakoniec sa pridá takto naplnená inštancia triedy `SelectContainer` do výsledného zoznamu.

```
List<SelectContainer1> f = new ArrayList<SelectContainer1>();
if(true){
  for (Person  person :  people){
    SelectContainer1 selectContainer7 = new SelectContainer1();
    selectContainer7.setMeno( person.getName());
    selectContainer7.setPriezvisko( person.getLastName());
    f.add(selectContainer7);
  }
}
```

Obrázok 3 – 16 Odpovedajúci zdrojový kód v čistej *Jave* pre 3 – 15

Nasledujúci a posledný prípad 3 – 17 je obdobný ako predchádzajúci, avšak s tým rozdielom, že nie sú uvedené premenné, ktoré sa majú nastavovať. Tu však je možné odvodiť názov nastavovanej premennej z celkového výrazu. Teda ak uvažujeme napríklad výraz `person.getName()`, tak odpovedajúca premenná, ktorá sa má nastaviť v rámci inštancií triedy `SelectContainer` bude mať názov `Name` a metódy na zapuzdrenie `setName` a `getName`.

```
var k = FROM person IN people
        SELECT new {person.getName() ,person.getLastName()};
```

Obrázok 3 – 17 Selekcia viacerých dát

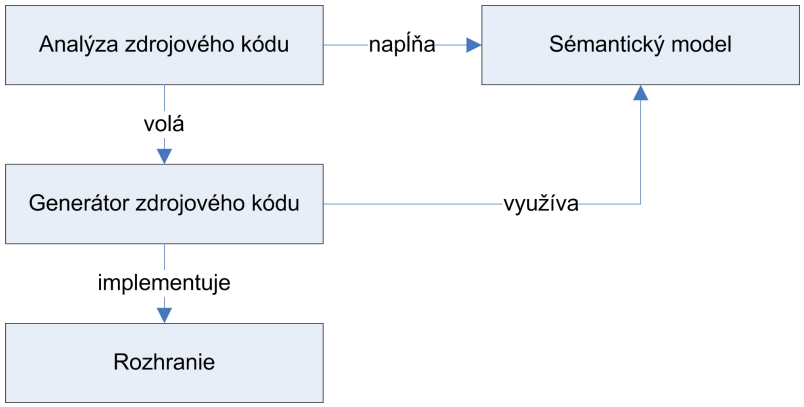
Odpovedajúci zdrojový kód v čistej *Jave* je možné vidieť na obrázku 3 – 18.

```
List<SelectContainer1> k = new ArrayList<SelectContainer1>();
if(true){
  for(Person  person :  people){
    SelectContainer1 selectContainer21 = new SelectContainer1();
    selectContainer21.setName(person.getName() );
    selectContainer21.setLastName(person.getLastName());
    k.add(selectContainer21);
  }
}
```

Obrázok 3 – 18 Odpovedajúci zdrojový kód v čistej *Jave* pre 3 – 17

3.3 Návrh architektúry riešenia

Celkové riešenie je možné rozdeliť na tri väčšie časti. Prvou časťou je definovanie sémantického modelu slúžiaceho na uchovanie podstatných údajov, ďalšia časť patrí analýze zdrojového kódu a posledná časť samotnému generovaniu. Ďalej je potrebné vytvoriť aj rozhranie definujúce metódy, ktoré sú potrebné pri generovaní zdrojového kódu. Toto rozhranie teda umožňuje rošírovanie sémantiky syntaxe. Každý generátor kódu implementuje jemu odpovedajúce rozhranie. Všetky analyzačné súbory, generatívne súbory, sémantické modely a rozhrania sú umiestnené v adresári `dsl/java/JaiqActions/`. Spolupráca jednotlivých častí je znázornená na obrázku 3 – 19.



Obrázok 3 – 19 Myšlienka rozloženia implementácie

4 Implementácia riešenia

4.1 Príprava

Celkovú prípravu je možné rozdeliť do troch krokov. Prvým je stiahnutie nástroja *ATS*, druhým je vytvorenie *Ant* spúšťacieho súboru pre nový projekt a tretím je definovanie súboru rovností. Po týchto úvodných činnostiach je možné pokračovať definovaním syntaxe a sémantiky nového rozšírenia jazyka *Java*. Posledným úkonom bude vygenerovanie samotného prekladača. Meno tohto projektu, resp. prekladacieho programu bude *Jaiq* (z angl. *JAva Integrated Query*).

Počiatočným krokom pri tvorbe rozšírenia pomocou nástroja *ATS* je jeho stiahnutie zo stránky `http://www.cs.utexas.edu/~schwartz/ATS/ATS`. Aktuálne sú k dispozícií dve verzie programu. Prvá je z roku 2011 a druhá z roku 2012. Pri tejto práci bola použitá verzia `ahead-v2011.05.03`. Pri tvorbe rozšírenia bolo postupované podľa ukážky postupu rozširovania syntaxe jazyka *Java* uvedeného na stránke `http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/ExtendJava.html`. Nástroj *ATS* je používaný pod operačným systémom *Linux*. Príkazy na generovanie kompilátora sa štandardne zadávajú z príkazového riadku. Na úpravu súborov je možné používať napríklad nástroj *Gedit*.

Po stiahnutí a rozbalení nástroja *ATS* je možné pozorovať takúto súborovú štruktúru:

- build
  - bin - spustiteľné programy na prácu s jazykom
  - lib - knižnice pre programy *ATS*
- docs - dokumentácia a príklady
- dsl

  - java
    - Java - definovaná sémantika *Javy*
    - JavaGram - definovaná syntax *Javy* v jazyku *Bali*
  - kernel - jadro umožňujúce prácu s abstraktnými stromami
- equations - obsahuje súbor rovností definujúci jazykové úrovne
- miscellaneous - rôzne pomocné súbory
- README.html
- setperms - nastavuje práva k súborom programu *ATS*

Pre prácu s nástrojom je potrebné nastavenie prístupových práv. Toto nastavenie zabezpečuje súbor `setperms`. Ten je nutné spustiť na konzole s právami administrátora v adresári, kde je vidieť adresár `ahead-v2012.01.05`.

```
./ahead-v2012.01.05/setperms
```

Ďalším krokom pri tvorbe jazyka alebo jazykového rozšírenia je nastavenie *Ant XML* súboru. Po spustení terminálu je potrebné nastaviť sa do adresára `ahead` (príkazom `cd`), kde nižšie uvedeným príkazom sa vytvorí spúšťací súbor *XML* pre *Ant*. Tu je potrebné iba zadať vlastný názov projektu uvedený pod parametrom `-Dproject.name` na `Jaiq`. Výstupom tohto príkazu bude *Ant XML* súbor `Jaiq.xml`.

```
ant -f docs/Configure.xml  -Dproject.name="Jaiq"
```

Nasledujúcim krokom je vytvorenie súborovej štruktúry nového rozšírenia. Tú je potrebné vytvoriť v adresári `ahead/dsl/java`. Je nutné vytvoriť dva adresáre. Prvý pre súbor definujúci syntax s názvom `JaiqGram`, druhý pre súbory definujúce význam syntaxe s názvom `JaiqActions`. Na vytvorenie adresárov pod systémom *Linux* je možné použiť príkaz `mkdir`, kde ako parameter sa uvedie názov adresára, ktorý sa má vytvoriť.

Tretím krokom je definovanie vrstiev v súbore rovností. Ten je potrebné vytvoriť v adresári `ahead/equations`. Názov súboru pozostáva z názvu projektu a prípony `equation` a teda `Jaiq.equation`. Obsah tohto súboru je uvedený na obrázku 4 – 1. Jadro (z angl. kernel), *Java* syntax a *Java* sémantiku poskytuje nástroj *AHEAD*. Jadro obsahuje triedy pre stavbu a prechádzanie syntakticky analyzovaného stromu. Vrstvy `dsl/java/JavaGram` a `dsl/java/JaiqGram` sú gramatické vrstvy. `dsl/java/Java` a `dsl/java/JaiqActions` sú sémantické vrstvy. Vrstvy `build/dsl/java/JavaGram` a `build/dsl/java/JaiqGram` sú použité pre program *applybali2jak*. Jeho zavolanie zabezpečuje súbor `Jaiq.xml`, ktorý bol vygenerovaný v prvých krokoch.

```
dsl/kernel
dsl/java/JavaGram
build/dsl/java/JavaGram
dsl/java/Java
dsl/java/JaiqGram
build/dsl/java/JaiqGram
dsl/java/JaiqActions
```

Obrázok 4 – 1 Obsah súboru `Jaiq.equation`

Tu je potrebné dodržať poradie aby gramatická vrstva *Javy* (`dsl/java/JavaGram`) bola pred gramatickou vrstvou nového rozšírenia (`dsl/java/JaiqGram`). Obdobne sémantická vrstva *Javy* (`dsl/java/Java`) má byť pred sémantickou vrstvou nového rozšírenia (`dsl/java/JaiqActions`). Preto je možné tiež napísať ekvivalent uvedený na obrázku 4 – 2, kde sú jednotlivé syntaktické a sémantické vrstvy „pri sebe“. V tejto práci je však použitá možnosť 4 – 1, pretože to predstavuje lepšiu orientáciu voči novo pridávanému rozšíreniu. Je v tom možné vidieť, že najprv sa definujú položky pre *Javu* ako hostovský jazyk a potom sú definované položky pre pridávané rozšírenie.

Toto sú základné úkony, ktoré je potrebné vykonať pred definovaním syntaxe a sémantiky a pred generovaním samotného prekladača.

```
dsl/kernel
dsl/java/JavaGram
dsl/java/JaiqGram
build/dsl/java/JavaGram
build/dsl/java/JaiqGram
dsl/java/Java
dsl/java/JaiqActions
```

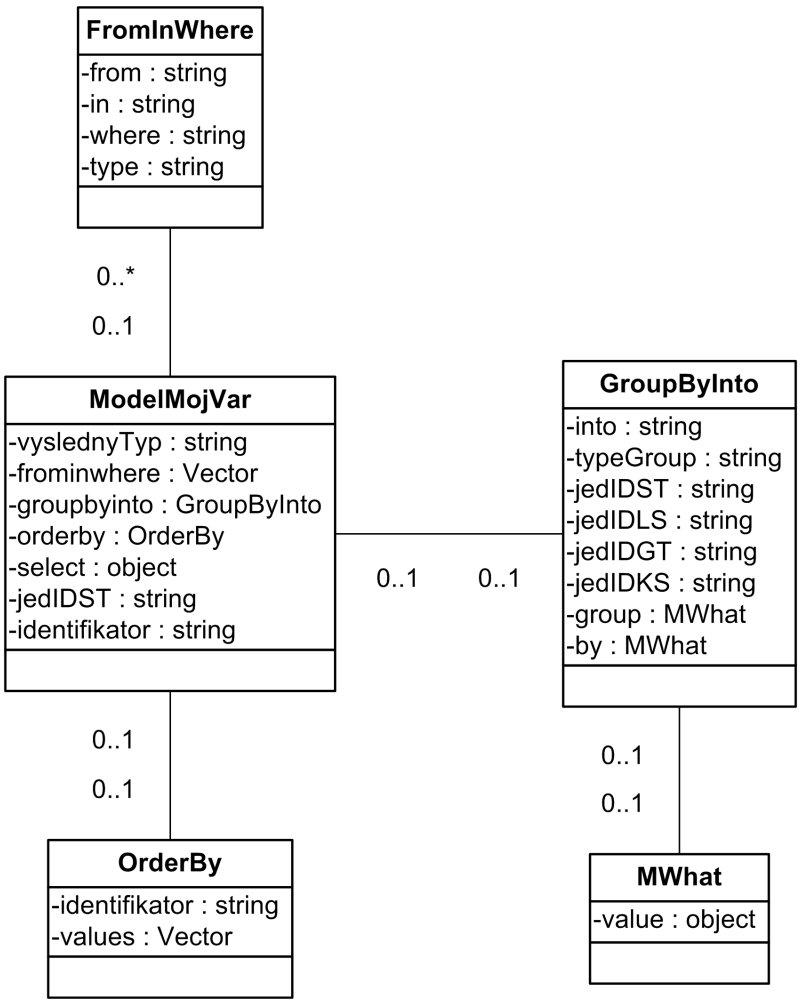
Obrázok 4 – 2 Alternatíva obsahu súboru Jaiq.equation

4.2 Sémantický model

Tento model slúži na uchovanie dôležitých údajov. Tieto údaje sú napĺňané v programovej časti analýzy zdrojového kódu a predané do generovanej etapy. Príklad sémantického modelu, ktorý je použitý pri spracovaní príkazu na selekciu údajov (v súbore `MojVar`) je znázornený na obrázku 4 – 3. Na tomto obrázku nie sú pre jednoduchosť znázornené metódy umožňujúce zapuzdrenie. Obdobné sú aj sémantické modely ostatných analyzovaných častí. Triedy reprezentujúce model: `ModelMojVar`, `FromInWhere`, `GroupByInto`, `ModelProgram`, `MWhat`, `OrderBy`, `Simple`, `One`, `Two`, `Known`, `Unknown`, `Zoznam`.

4.3 Analýza zdrojového kódu

Táto časť slúži na naplnenie sémantického modelu potrebnými údajmi a na následné volanie generatívnej metódy. Analýza zdrojových kódov s novým rozšírením je spracovaná v súboroch `AST_Program`, `AST_Imports`, `MojVar`. Na analýzu sa použije trieda `Cursor`, umožňujúca prechádzanie uzlami *AST*. Príklad časti zdrojového kódu, ktorý analyzuje príkaz na selekciu je znázornený na obrázku 4 – 4. Ten spracúva údaje uvedené za klauzulami `FROM` a `WHERE`.



Obrázok 4 – 3 ModelMojVar

4.4 Generovanie výstupného zdrojového kódu

Samotné generovanie zdrojového kódu spočíva v spracovaní daného modelu vybratím podstatných údajov a ich aplikáciou na statický kód. Triedy zabezpečujúce generovanie zdrojového kódu: `AST_ProgramGen`, `MojVar_Gen`, `ForVarStm`, `AST_ImportsGen`. Z týchto spomenutých využívajú sémantický model iba prvé dve triedy. Dôvod je ten, že zvyšné triedy obsahujú natoľko jednoduché generovanie, že nie je potrebné pre nich vytvárať sémantický model. Triedy slúžiace na generovanie implementujú im odpovedajúce rozhranie. Toto rozhranie spolu so sémantickým modelom umožňuje ďalšie rozširovanie použiteľnosti nových príkazov (napr. podpora databáz). Hlavná

```
Integer zatvorky =0;
for (c.First(arg[1]); c.More(); c.PlusPlus()){
  if(c.node instanceof FromM ){
    zatvorky++;
    AstNode qnamef = NajdiQName(c.node);
    AstNode qnamei = NajdiQName(NajdiAIntoM(c.node));
    String typeList = HladajTyp(qnamei.toString());
    if(typeList.isEmpty()){
      System.err.println("Neexistuje zoznam s menom "+
      qnamei.toString());
      System.exit(0);
    }
    FromInWhere fiw = new FromInWhere(
qnamef.toString(), qnamei.toString(), typeList);
    model.AddFromInWhere(fiw);
    DNastavPremennu(qnamef,typeList);
  }else if(c.node instanceof WhereM ){
    zatvorky++;
    AstNode expression = NajdiExpression(c.node);
    Zoznam z = (Zoznam)model.GetFromInWhere();
    FromInWhere fiw = (FromInWhere) z.GetLast();
    fiw.SetWhere(expression.toString());
    z.SetLast(fiw);
    model.SetFromInWhere(z);
  }
}
```

Obrázok 4 – 4 Ukážka časti analýzy selektívneho príkazu

myšlienka všetkých generátorov je znázornená na príklade 4 – 5. Premenná `mp` reprezentuje model programu. Z tohto modelu sa získa zoznam všetkých premenných, ktoré bude obsahovať nová trieda. Jeho prechádzaním sa dopĺňajú údaje do reťazcov, ktoré sú v závere zapísané do výstupného súboru. V tomto prípade sa vytvárajú privátne premenné, ich zapuzdrenie, funkcie `hash` a `toString`. Obdobná myšlienka

generovania je použitá aj v ostatných generátoroch zdrojového kódu.

```
Zoznam premenne = mp.getPremenne();
if(premenne.isEmpty()) return;
for(int i = 0; i < premenne.GetIndex(); i++){
  String premenna = (String) premenne.Get(i);
  privateObjects+="\tprivate Object "+premenna+";\n";
  getSetrs+="\tpublic Object get"+premenna+"(){\n\t\treturn this."
  +premenna+";\n\t}\n";
  getSetrs+="\tpublic void set"+premenna+"(Object "+premenna
  +"){\n\t\tthis."+premenna+"="+premenna+";\n\t}\n";
  hash += "\tif(this.get"+premenna+"() != null){\n";
  hash += "\t\tres += \" \" + this.get"+premenna+"().toString();\n\t}\n";
  metodaToString += "\tif(this.get"+premenna+"() != null){\n";
  metodaToString += "\t\tres += \" \" + this.get"+premenna
  +"().toString();\n\t}\n";
}
```

Obrázok 4 – 5 Generovanie častí triedy `SelectContainer`

4.5 Pridávanie importov

V súbore `AST_Imports` je napísaná logika pre analýzu importov. Pri generovaní vznikajú zdrojové kódy, ktoré vyžadujú použitie špecifických *Java* balíkov. Takýmto balíkom môže byť aj balík `java.util` zabezpečujúci potrebné kolekcie (napr. `ArrayList`, `HashMap`,atď). V prípade, ak nie je súčasťou importu, tak je potrebné ho doplniť, keďže riešenie selekcie dát je od neho závislé. Takúto analýzu zdrojového kódu vykonáva práve trieda `AST_Imports`. Tá v prípade, ak je potrebný nejaký balík, tak ho pridá do zoznamu potrebných importov a tento zoznam je v závere použitý na generovanie v súbore `AST_ImportsGen`.

4.6 SelectContainer

Pretože nie vždy je možné určiť typ pre daný novo vytváraný zoznam, tak je potrebné vytvoriť štruktúru, ktorá by bola schopná reprezentácie tohto typu. Takisto táto štruktúra by mala slúžiť ako pomocná štruktúra pre zoskupovanie údajov za použitia príkazu `GROUP`. Tu je možné sa rozhodnúť, či použiť dve štruktúry, jednu pre výsledný zoznam a druhú pre zoskupovanie, alebo ich spojiť. V tejto diplomovej práci bol použitý druhý spôsob a to kvôli zjednodušeniu manipulácie s touto štruktúrou. Táto trieda obsahuje potrebné privátne premenné, ich zapuzdrenie, funkcie pre porovnávanie (`hashCode`, `equals`) a pre výpis hodnôt premenných (`toString`). Vytvára sa na konci *Java* súboru a jej názov pozostáva zo slova `SelectContainer` a pridaného identifikačného čísla. Ukážku triedy, ktorá uchováva hodnoty premenných `key`, `Name` a `LastName` je možné vidieť na obrázkoch 4 – 6 4 – 7 4 – 8 4 – 9. Položka `key` sa vytvára vždy, ak súbor obsahuje aspoň jeden príkaz na zoskupovanie údajov za použitia klauzuly `GROUP`. Keďže v okamihu generovania sa nedá určiť typ premenných, tak bol použitý univerzálny typ `Object`. Názvy premenných, ktoré sa majú generovať sa získavajú v analyzačnom súbore `AST_Program`. Ten ich pomocou sémantického modelu `ModelProgram` poskytne generátoru, ktorý sa nachádza v súbore `AST_ProgramGen`.

V analýze zdrojového kódu sa hľadajú uzly typu `WhatMN`. Tento typ uzla sa môže nachádzať za klauzulami `GROUP`, `BY` a `SELECT` a umožňuje výber viacerých argumentov za pomoci kľúčového slova `new` a vlnitých zátvoriek. Jednotlivé argumenty výberu sú oddelené čiarkou. Z takéhoto kódu je možné odvodiť názvy premenných. Uvažujme takúto časť zdrojového kódu: `new {Meno = person.getName(), Priezvisko = person.getLastName()}`. Názvy privátnych premenných budú `Meno` a `Priezvisko`.

Druhým prípadom je, ak nie sú uvedené položky, ktoré sa majú nastaviť a časť príkazu vyzerá nasledovne: `new {person.getName(), person.getLastName()}`. Tu sa ako názvy premenných vyberú `Name` a `LastName`.

Ďalšou možnosťou ako môže byť pridaná nová premenná do triedy `SelectContainer` je ak sa nepodarí zistiť výsledný typ pri selekcií dát. V tomto prípade je riešenie obdobné ako pri selekcií viacerých údajov s tým rozdielom, že tu získava názov novej premennej iba z jedného výrazu.

Názvy zapuzdrovacích metód sú vytvárané z mien privátnych premenných. Ak metóda slúži na nastavenie privátnej premennej, tak jej meno bude pozostávať zo slova `set` a z názvu danej premennej. Názov metódy, ktorá získava hodnotu, bude obdobný ako v predošlom prípade, avšak namiesto slova `set` sa použije slovo `get`. Prístup k hodnotám týchto premenných je za pomoci zapuzdrenia.

Myšlienka metódy `hash` spočíva v zreťazení textových hodnôt všetkých premenných, ktoré sú nastavené. Po zreťazení hodnôt sa získa hash hodnota výsledného reťazca. Táto metóda je dôležitá pri porovnávaní dvoch inštancií a teda aj pri použití zoraďovacej klauzuly `ORDERBY`.

Metóda s názvom `toString` vracia reťazec, ktorý je zložený z privátnych hodnôt tejto triedy. Myšlienka je obdobná ako pri metóde `hash`, avšak s tým rozdielom, že na konci sa už nezískava hash hodnota, ale vracia sa iba zreťazená textová hodnota.

Ďalšou dôležitou metódou, ktorá slúži na porovnanie dvoch tried je metóda s názvom `equals`. Tá najprv porovná aktuálnu inštanciu triedy, s inštanciou triedy uvedenou v argumente metódy. V prípade, ak ich názvy tried sa zhodujú, tak prejde na porovnávanie za pomoci funkcie `hash`. Táto metóda je zaujímavá tým, že neobsahuje žiadne dynamicky generované časti a je teda pevne daná.

4.7 Zistenie dátového typu pre danú premennú

Pri vytváraní nového zoznamu je často potrebné zistiť dátový typ pre daný identifikátor. Na tento účel je vytvorený statický zoznam pozostávajúci z názvov premenných a im odpovedajúcich dátových typov. Tento zoznam sa napĺňa v priebehu prechá-

```
class SelectContainer1 {
  public SelectContainer1(){}
  private Object key;
  private Object Name;
  private Object LastName;

  public Object getkey(){ return this.key; }
  public void setkey(Object key){ this.key=key; }
  public Object getName(){ return this.Name; }
  public void setName(Object Name){ this.Name=Name; }
  public Object getLastName(){ return this.LastName; }
  public void setLastName(Object LastName){
 this.LastName=LastName; }
```

Obrázok 4 – 6 Príklad vygenerovanej triedy `SelectContainer` jej privátne premenné a zapuzdrovacie metódy

```
  @Override
  public int hashCode() {
    String res="";
    if(this.getkey() != null)
      res += " " + this.getkey().toString();
    if(this.getName() != null)
      res += " " + this.getName().toString();
    if(this.getLastName() != null)
      res += " " + this.getLastName().toString();
    return res.hashCode();
  }
```

Obrázok 4 – 7 Príklad vygenerovanej triedy `SelectContainer` metóda `hashCode`

dzania jednotlivých príkazov na selekciu dát a obsahuje jeho identifikátor (medzi kľúčovým slovom `var` a znamienkom `=`) a výsledný dátový typ. Tento zoznam je k dispozícií aj pri ďalšom spracovaní príkazov na selekciu dát a teda umožňuje prechádzať aj už predtým vyberané dáta. Inými slovami, ak máme dva príkazy na selekciu dát, tak v tom druhom príkaze je možné použiť identifikátor prvej selekcie údajov

```
  @Override
  public String toString() {
    String res = new String("");
    if(this.getkey() != null)
      res += " " + this.getkey().toString();
    if(this.getName() != null)
      res += " " + this.getName().toString();
    if(this.getLastName() != null)
      res += " " + this.getLastName().toString();
    return res;
  }
```

Obrázok 4 – 8 Príklad vygenerovanej triedy `SelectContainer` metóda `toString`

```
  @Override
  public boolean equals(Object obj) {
    if(this.getClass().equals(obj.getClass())){
      if(this.hashCode() == obj.hashCode()){
        return true;
      }
    }
    return false;
  }
}
```

Obrázok 4 – 9 Príklad vygenerovanej triedy `SelectContainer` metóda `equals`

a priamo zistiť jeho odpovedajúci dátový typ.

Ďalším obdobným zoznamom je zoznam, ktorý sa napĺňa na začiatku spracovania príkazu na selekciu dát. Tento zoznam slúži na uloženie dočasných názvov premenných a im odpovedajúcich dátových typov. Tieto dočasné názvy premenných sú uvedené za klauzulou `FROM` a keďže v jednom príkaze na selekciu dát ich môže byť viacero, tak je dobré mať takýto zoznam.

Uvažujme časť príkazu: `FROM person IN people`. Pre dočasnú premennú `person` je potrebné nájsť dátový typ. Ten možno odvodiť zo zoznamu `people`. Pri hľadaní

tohto dátového typu sa prehľadáva trieda a metóda v ktorých je umiestnený príkaz na selekciu údajov. Pri prezeraní metódy a potom aj triedy sa hľadá zhoda v mene prechádzaného zoznamu s hľadaným zoznamom. Teda pre tento prípad sa hľadá zoznam s identifikátorom `people`. V prípade, ak sa takýto zoznam nájde, tak je potrebné z neho vybrať dátový typ. V tomto prípade je tento typ `Person`. Do výsledného zoznamu sa uloží názov premennej `person` a k nej odpovedajúci dátový typ `Person`. Následne je možné vytvoriť metódu, ktorá prehľadáva oba zoznamy a hľadá v nich dátový typ pre odpovedajúcu premennú. Takto zistený dátový typ je ďalej možné použiť ako výsledný typ zoznamu, alebo v prípade metódy na zoraďovanie údajov.

4.8 Generovanie jedinečného identifikátora

Aby novo pridávané identifikátory neboli v konflikte s už existujúcimi identifikátormi, tak je potrebné zabezpečiť ich správne generovanie. Takéto identifikátory sú použité pri identifikácií zoskupovacej skupiny, porovnávacej triedy, pomocných zoznamov a pod. Funkcia zabezpečujúca ich generovanie má názov `ZiskajJedinecneID` a ako vstupné argumenty potrebuje uzol aktuálnej triedy, uzol aktuálnej metódy a počiatočný názov nového identifikátora. Túto funkciu je možné nájsť v súbore `MojVar` zabezpečujúcom analýzu zdrojového kódu. Celkový názov identifikátora bude pozostávať z počiatočného názvu a k nemu pridané identifikačné číslo. Toto číslo je statické v rámci triedy a pri každom použití sa jeho hodnota inkrementuje. V prípade, ak už existuje identifikátor, ktorý je rovnaký ako generovaný celkový názov identifikátora, tak sa hodnota identifikačného čísla inkrementuje a znova sa otestuje jedinečnosť. Tento postup sa opakuje dokiaľ existuje zhoda z existujúcimi identifikátormi. Test zhody sa zisťuje porovnávaním generovaného identifikátora a identifikátorov v rámci uzla aktuálnej triedy a uzla aktuálnej metódy.

5 Záver

Výsledkom tejto práce je stručný opis existujúcich možností rozširovania jazyka *Java* a vytvorené vlastné rozšírenie. V práci boli opísané nástroje *ATS*, *JSE*, *Maya*, *SugarJ* a *Lombok*. Po ich analýze sa na tvorbu rozšírenia použil nástroj *ATS*. Riešenie splnilo daný cieľ a v konečnom výsledku prekladací program umožňuje správny preklad zdrojového kódu s rozšírením do zdrojového kódu v čistej *Jave*. Cieľom nebolo pridanie celej syntaxe rozšírenia *LINQ*. Pridané kľúčové slová čiastočne odpovedajú rozšíreniu *LINQ* v jazyku *C#*. Vytvorené riešenie je možné používať na selekciu údajov zo zoznamu dát.

Analýza nástrojov umožňujúcich rozširovanie jazyka Java bola zameraná hlavne na jednoduchosť ich používania. Výhodou bolo ak bol nástroj priamo určený na rozširovanie jazykov. Preto bol vybraný ako nástroj na tvorbu rozšírenia práve *ATS*, ktorého ďalšou kladnou vlastnosťou bolo používanie *AST*.

Po vybratí nástroja na tvorbu rozšírenia jazyka Java nasledovalo definovanie syntaxe. Tu bola použitá jazyková špecifikácia *Bali*. Ďalším krokom bolo definovanie sémantických modelov, ktoré zabezpečia uchovanie a predávanie dát. Nasledovalo vytvorenie analyzačných súborov, ktoré sémantické modeli naplnia. Posledným krokom, pri ktorom bolo potrebné písať zdrojový kód, bolo vytvorenie súborov zabezpečujúcich generovanie zdrojového kódu na základe sémantického modelu. Potom bolo možné vygenerovať prekladač jazyka s rozšírením. Takto vygenerovaný prekladač je možné používať na preklad s textom s rozšírením do kódu bez rozšírenia.

Výhodou riešenia je ľahšia organizácia údajov a ľahšia manipulácia s dátami. Oproti iným podobným riešeniam je zdrojový kód napísaný v jazyku rozšírenia omnoho lepšie čitateľný. Za kladnú vlastnosť možno považovať aj dôkladné spracovanie syntaktických chýb.

Nevýhodou riešenia je absencia automatického dopĺňania zdrojového kódu, ktorú programátori často používajú. Zdrojový kód je potrebné písať mimo programova-

cieho prostredia, avšak po preložení zdrojových kódov je možné tieto výstupné texty importovať do vývojového prostredia. Ďalšou nevýhodou môže byť potreba dvojitého prekladu. Teda najprv sa prekladá za pomoci programu vytvoreného v práci a potom za pomoci javac prekladača. Tento problém je možné riešiť napríklad použitím jedného spúšťacieho súboru (*bash*), ktorý by vykonal uvedené preklady v danom poradí.

Výsledné riešenie je možné rozširovať vďaka pridanému rozhraniu a sémantickým modelom. Riešenie je možné rozšíriť o spracovanie údajov uložených v *XML* súboroch, databázach, a pod. V tomto prípade by bolo potrebné jemne upraviť zdrojové kódy zabezpečujúce analýzu zdrojového textu a napísať odpovedajúce generátory zdrojových dát.

Zoznam použitej literatúry

[1] Ahead Tool Suite website [online]. 2008. [cit. 2011-5-1]. Dostupné na internete: <http://www.cs.utexas.edu/~schwartz/ATS/fopdocs>.

[2] ASM website [online]. 2012. [cit. 2011-5-3]. Dostupné na internete: <http://asm.ow2.org>.

[3] BAKER, Jason - HSIEH, Wilson: Maya: Multiple-Dispatch Syntax Extension in Java. In: PLDI '02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. 2002. ISBN:1-58113-463-0.

[4] BATORY, Don - SARVELA, Jacob Neal - RAUSCHMAYER: Scaling Step-Wise Refinement. In: IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. 2004, č.6, s.355-371.

[5] BATORY, Don : Feature-Oriented Programming and the AHEAD Tool Suite. In: ICSE '04 Proceedings of the 26th International Conference on Software Engineering. 2004. ISBN:0-7695-2163-0.

[6] Custom AST transformations with Project Lombok website [online]. [cit. 2011-4-5] . Dostupné na internete: <http://www.ibm.com/developerworks/java/library/j-lombok>.

[7] DataNucleus website [online]. 2012. [cit. 2011-5-10]. Dostupné na internete: <http://www.datanucleus.org>.

[8] Datastore website [online]. 2012. [cit. 2011-6-1]. Dostupné na internete: <https://developers.google.com/appengine/docs/java/gettingstarted/usingdatastore>.

[9] ERDWEG, Sebastian at al.: SugarJ: Library-based Syntactic Language. In: OOPSLA '11 Proceedings of the 2011 ACM international conference on Object

oriented programming systems languages and applications. 2011. ISBN: 978-1-4503-0940-0.

[10] Getting Started with the Annotation Processing Tool [online]. [cit. 2011-4-10]. Dostupné na internete: <http://docs.oracle.com/javase/6/docs/technotes/guides/apt/GettingStarted.html>.

[11] Iciql website [online]. 2012. [cit. 2011-5-25]. Dostupné na internete: <http://iciql.com>.

[12] JaQue website [online]. 2012. [cit. 2011-5-5]. Dostupné na internete: <http://code.google.com/p/jaque>.

[13] Java Syntactic Extender website [online]. [cit. 2011-4-5]. Dostupné na internete: <http://jse.sourceforge.net>.

[14] KIMBERLIN, Michael: Reducing Boilerplate Code with Project Lombok. [pdf] [cit. 2011-4-13]. Dostupné na internete: <http://jnb.ociweb.com/jnb/jnbJan2010.html>.

[15] Oracle Annotations website [online]. [cit. 2011-3-25]. Dostupné na internete: <http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html>.

[16] PETERSSON, Kent: Syntax and Semantics of Programming Languages. Göteborg: University of Göteborg and Chalmers. 1990. 92 s.

[17] PIALORSI, Paolo - RUSSO, Marco: Introducing Microsoft LINQ. Redmond: Microsoft Press, 2007. 203 s. BPN X13-68390.

[18] PORUBÄN, Jaroslav: Anotácie [online]. [pdf]. [cit. 2011-3-3]. Dostupné na internete: <http://hornad.fei.tuke.sk/~poruban/magsa/M5slide.pdf>.

[19] Project Lombok website [online]. [cit. 2011-4-15]. Dostupné na internete: <http://projectlombok.org>.

[29] Project Lombok: Creating Custom Transformations [website]. [cit. 2011-4-23]. Dostupné na internete: <http://notatube.blogspot.com/2010/12/project-lombok-creating-custom.htm>.

[21] QueryDSL website [online]. 2012. [cit. 2011-5-3]. Dostupné na internete: <http://iciql.com>.

[22] SARVELA, Jacob Neal: The Bali Language [pdf]. 2003. [cit. 2012-1-19]Dostupné na internete: <http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/bali.pdf>.

[23] SBQL4J website [online]. 2012. [cit. 2011-6-9]. Dostupné na internete: <http://code.google.com/p/sbql4j>.

[24] Stratego website [online]. [cit. 2011-4-5]. Dostupné na internete: <http://strategoxt.org>.

[25] Stack-Based Architecture (SBA) and Stack-Based Query Language (SBQL) website [online]. 2008. [cit. 2011-5-30]. Dostupné na internete: <http://sbql.pl/overview>.

[26] SugarJ website [online]. [cit. 2011-5-5]. Dostupné na internete: <http://www.uni-marburg.de/fb12/ps/research/sugarj>.

[27] Trail: The Reflection API [online]. [cit. 2011-4-10]. Dostupné na internete: <http://docs.oracle.com/javase/tutorial/reflect>.

[28] VISSER, Joost - SCHEERDER,Jeroen: A Quick Introduction to SDF. [pdf]. 2000. Dostupné na internete: <ftp://ftp.stratego-language.org/pub/stratego/docs/sdfintro.pdf>.

[29] Vnútorné uzamknutie a synchronizácia [website]. [cit. 2011-4-20]. Dostupné na internete: <http://java.mrazovci.eu/sk/node/95>.

[30] ZINGARO, Daniel: Modern Extensible Languages. [pdf]. McMaster Uni-

versity, 2007. [cit. 2011-4-19]. Dostupné na internete: <http://www.danielzingaro.com/extensible.pdf>.

Zoznam príloh

Príloha A Používateľská príručka

Príloha B Systémová príručka

Príloha C Zdrojové kódy na CD médiu