Reverse-engineering EBNF Grammars into MOF Metamodels

Project Work

Submitted by:
LIU Yao
Liuyao1981918@hotmail.com
IMT
Matriculation Number:
27497

Supervised by:

Prof. Dr. Ralf Möller
STS - TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Hamburg, Germany
2005-06-30

Declaration of Originality

I declare that:

This work has been prepared by myself,
all literally or content-related quotations from other sources are clearly pointed out,
and no other sources or aids than the ones that are declared are used.

Hamburg, 30.06.2005

Abstract

Semantic Web will probably make important influence on web technologies. Description logics is the fundamental mechanism for describing the semantics embedded in the web. Therefore, processing description logics is attracting more and more attention of researchers. Prof. Dr. Volker Haarslev, Kay Hidde, Prof. Dr. Ralf Möller and Michael Wessel have developed a description logic inference system, which accepts queries written in nRQL, a query language specially designed for Description Logic. But the client side of this system is not so complete as needed. Some preprocessing in the client on queries is expected, before they are sent as requests.

This report describes a student project focusing on building a prototype for transforming an EBNF grammar (e.g. that of nRQL) to an object-oriented metamodel, so that later parsing a document conforming to that grammar (in the example, a document containing nRQL queries) results in objects instantiated from the OO metamodel.

This report explains the approach for transformation, introduces two important tools used in this project, and explains in detail how the prototype was built. When building the prototype, comparison between two implementation means are involved, some complex problems encountered are addressed with the solution finally given, and some important features of certain parts of the prototype are clarified also. At the end, conclusions are given.

1 Introduction

1.1 Motivation

World Wide Web is developing very fast in recent years, mostly acting as a medium of documents for people to publish and obtain data. But it almost does not provide information that can be manipulated automatically. The need of machine-processable data embedded in web pages made the technology Semantic Web come in, which is a mesh of information linked up with each other to be easily processed by machines. This is an efficient way of representing data on World Wide Web, just like a globally linked database.

To achieve this, schemas and ontologies were introduced to add semantics to URIs and data which was hidden in the web pages, so that the semantic data could be retrieved by following the rules for reasoning about data logically and the hyperlinks to definitions of key terms.

One of the ontology languages, Web Ontology Language (OWL), is a syntactic variant of a well-known and very expressive description logic, which can be used to explicitly represent the meaning of terms in vocabularies and the relationships between those terms. This representation of terms and their interrelationships is called an ontology. Then, we could get the point that, a description logic that ontologies are based on is the essential in representing data semantics.

Therefore, dealing with description logics has become an important task in the research and development of Semantic Web technology. One of the efforts in front of our eyes is RacerPro system, which is being developed by Prof. Dr. Volker Haarslev, Kay Hidde, Prof. Dr. Ralf Möller and Michael Wessel. RacerPro system is a description logic inference system, which implements a highly optimized tableau calculus for a very expressive description logic SHIQ that is the base of OWL DL, and offers reasoning services as well. *[1]*

The standard RacerPro acts like a face-less back-end server， and is therefore sometimes referred to as the "kernel application". Correspondingly, RacerPro needs the client side, by which humans can interact with RacerPro. Typically all interaction with RacerPro takes place via network protocols like HTTP (DIG) or Racer native commands (over TCP/IP). We also can implement our own client system using LRacer or JRacer as Lisp or Java libraries to the Racer native command set, but we have to provide all necessary functionallity as well as the user interface yourself. *[1]*

The existing client systems of RacerPro are RacerPorter using LRacer and RICE using JRacer. (Fig 1.1) By the two client systems, the description logic queries can be built in graphics, described in nRQL (new Racer Query Language, a high level query language designed for Racer queries) and sent to RacerPro to process. (Fig 1.2) *[1]*

However, there is no syntax checking in the client systems for the grammar correctness of queries in nRQL. It is a waste for RacerPro to reason grammatically wrong queries. Furthermore, sometimes we do not need so complex client user interfaces as the two client systems mentioned above, simple console for input and output is enough and would be better if it is convenient to facilitate some other development, integrated in the Eclipse IDE.

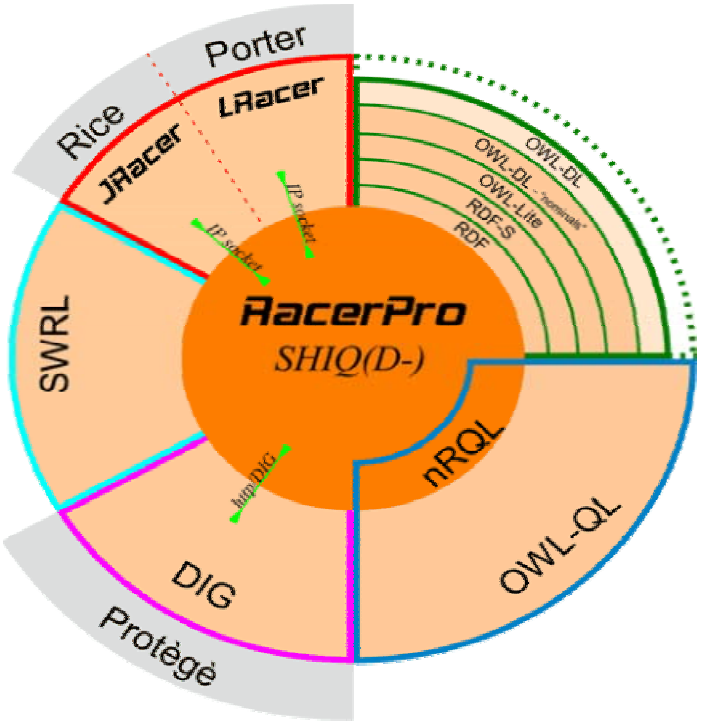

Fig 1.1 Structure of RacerPro

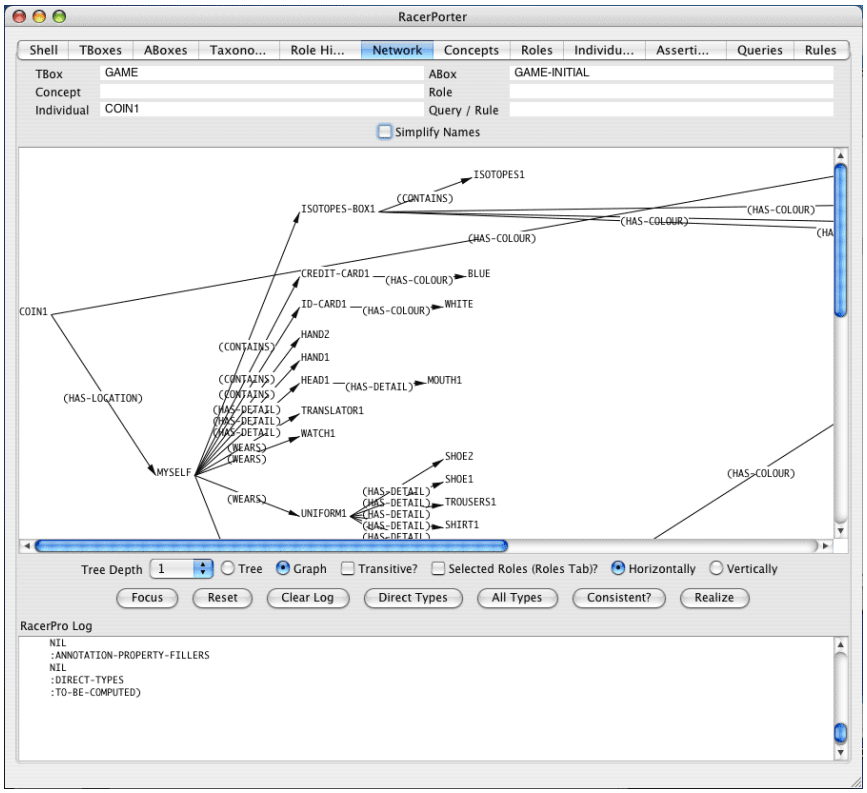

Fig 1.2 RacerPorter

1.2 Objectives and Goal of this project

What we want to achieve is to build a prototype that can analyze syntax of any queries obeying the specification of a certain simple language or expression. This prototype will be enriched in the future to check queries against the grammar of nRQL before the queries are sent to RacerPro. Moreover, this prototype without complex user interface is expected to be integrated into the Eclipse IDE.

The objectives of this project are:

1. Investigate EBNF for describing grammars of any languages;

2. Investigate an important paper “A Metamodel for SDL-2000 in the Context of Metamodelling ULF”, and summarize the approach addressed in it;

3. Get familiar with ANTLR and Octopus, learn to write a .g file and a .uml file for a prototype;

4. Design and construct a workflow diagram for the prototype;

5. Implement the prototype;

6. Encapsulate the prototype as a plug-in of Eclipse.

2 Metamodelling of Syntax

Today, the syntax of many languages is defined using context-free grammars. But these syntax definitions suffer from some drawback, therefore, metamodel is introduced to solve the problem. In this chapter, we will focus on metamodelling of the syntax of the existing languages. Firstly, EBNF is introduced simply in Section 2.1. Then in Section 2.2 to be discussed is why metamodel should be used. Later comes the core of this chapter. Section 2.3 presents how meta-metamodels for EBNF grammars are built. And how to transform from EBNF grammars to metamodels automatically is explained in Section 2.4. This is the essential aspect of this student project.

2.1 What is EBNF?

EBNF is Extended Backus-Naur Form, which is a formal mathematical way to describe a language. EBNF is used to formally define the grammar of languages. The grammar defined using EBNF consists of a collection of grammatical rules, and each rule is devided by ":=" into two parts, the left side is called NonTerminal, whereas the right side, which is going to replace the left side, can be a collection or sequence of symbols, NonTerminals or Terminals, in some relationships with the assistance of some special signs.

There is a real example, which defines a grammar for simple arithmetic calculations:

```
expr  :=  proexpr ((PLUS | MINUS) proexpr)* SEMI

proexpr  :=  powpr ((MUL | DIV) powpr)*

powpr    :=  atom (POW atom)?

atom     :=  INT

MUL      :=  '*'

DIV      :=  '/'

PLUS     :=  '+'

MINUS    :=  '-'

POW      :=   '^'

SEMI     :=  ';'

INT      :=  ('0'..'9')+
```

In this example, each line is a rule. The symbols on the left side of rules are NonTerminal, because they must be replaced by the right side of rules, e.g. *proexpr* must be replaced by *powpr ((MUL | DIV) powpr)**,etc. And Terminals are the strings that are not replaced by other Terminals or NonTerminals, and actually they terminate the replacement process, e.g. *'/'*, *'+'*, *'*'* etc. In the rule *proexpr := powpr ((MUL | DIV) powpr)**, the '|' between *MUL* and *DIV* is used to present that *MUL* and *DIV* are alternatives, either of which is combined with *powpr*.

We may notice some special signs used in this example, such as * , *?* and + (different from '*', '+', which are Terminals). The three special signs (operators) are the difference between EBNF and BNF.

- *? : which means that the symbol (or group of symbols in parenthesis) to the left of the operator is optional (it can appear zero or one time);*
- ** : which means that something can be repeated any number of times (and possibly be skipped altogether);*
- *+ : which means that something can appear one or more times. [2]*

Therefore, the rules with the operators above should be easily understood. For example, we can give an instance for *proexpr ((PLUS | MINUS) proexpr)**, 5 + 3 + 2, here *PLUS proexpr* appears twice, this is what operator * allows for.

EBNF is widely used to define the formal grammar of languages, because it has two advantages: there can be no ambiguity on the defined syntax of a target language, and it makes it much easier to make compilers, because the parser for the compiler can be generated automatically with a compiler-compiler like ANTLR, which we will introduce in a later chapter.

By now, main concepts of EBNF have been introduced. It seems that EBNF is very competent when defining the syntax of languages, why do we need metamodel, and what are the benefits if using metamodel? It is better move to next section to find out the answers.

2.2 Why Metamodel?

A metamodel is a model that explains a set of linguistic models. *[3]* Metamodel and Context free grammars are thought of as main methodologies to define coordinated syntax definitions.

As we mentioned in Section 2.1, EBNF syntax is developed to specify concrete language syntax without ambiguity. However, EBNF grammars do not provide the means for rule refinement or generalization, and also do not allow modularization. It is impossible in EBNF grammars to refine rules to form generalization hierarchies or to set up logical structures by using a namespace mechanism or something like that. *[4]*

Therefore, some benefits of object-oriented languages can not be taken from an EBNF grammar based specification, which we can call syntax-oriented. For example, generalization, and thus refinement and reuse, as well as logical structures through mechanisms like namespaces and packages, are very important concepts in language specifications and object-oriented development. *[4]* With these features, not only can the

words of a language be specified, but the internal structure of the described language can be defined. This allows a natural evolution of language specifications and easy tool development. Obviously, syntax-oriented method even is straightforward to describe a language syntax definition, but it can not provide some useful and important features that object-oriented method have, thus it is somehow not very suitable when we want to involve language syntax manipulation in an object-oriented development environment.

In contrast, as object-oriented modelling platforms, metamodelling architectures provide a generalization mechanism, so that metamodel can contain inheritance hierarchies of concepts. And the metamodelling language is required to be object-oriented. Now, we know object-oriented metamodels are fit for migrating manipulations on pure EBNF based language syntax onto object-oriented platforms.

2.3 Building EBNF Metamodel

After clarifying the potential of metamodelling for language specification, we faced the problem of providing this metamodelling features to already existing, grammar based language specifications. Here a method is proposed to solve this problem.

In order to adopt the essence of model-driven software engineering, we need an source model, a target model, and the rules for transforming from the origin model to the target model. In this case, we already have the source model, which is the EBNF grammars defining the syntax of computer languages. The target model is the metamodel for EBNF grammars. Then the transformation rules are designed. With the three parts, the transformation is required to be automatically carried out. That can avoid many errors due to human failure if it is executed manually.

The resulting metamodel should only use the existing meta-concepts provided by EBNF, as the automatic transformation is expected. Therefore, in order to easily understand the concepts in EBNF grammars and their relationships, it would be better first depict clearly the meta-concepts of EBNF and their relationships. Here a metamodel of EBNF, which is a meta-metamodel of EBNF grammars, is presented in Fig 2.1. *[4]*

The target model to achieve, corresponding to the origin model ---- EBNF grammars, is a metamodel, which should be object-oriented and can be represented in a UML diagram. Thus it is a good way to modify the meta-metamodel in Fig 2.1 and make it become the one that is easier for common metamodelling architectures and closer to the target object-oriented model, which means the concepts used in the two models, meta-metamodel in Fig 2.1 and the result of modification, would get related. Moreover, through the modification on the meta-meta level, we can deduce a mapping from the meta-concepts of EBNF to the object-oriented concepts used in object-oriented metamodels. Then the transformation rules created through the mapping is for the meta-level of all EBNF grammars, so that it would be the base of the automatic EBNF grammars to object-oriented metamodels conversion. *[4]*

The modification made to the model in Fig 2.1 should use concepts out of the concept space of object-oriented metamodelling, as long those concepts can replace the respective grammar concepts isomorphicly. In Fig 2.1, the grammar concepts Symbol, Terminal, NonTerminal, Rule and Expression are all basic elements in EBNF and they represent atomic entities in

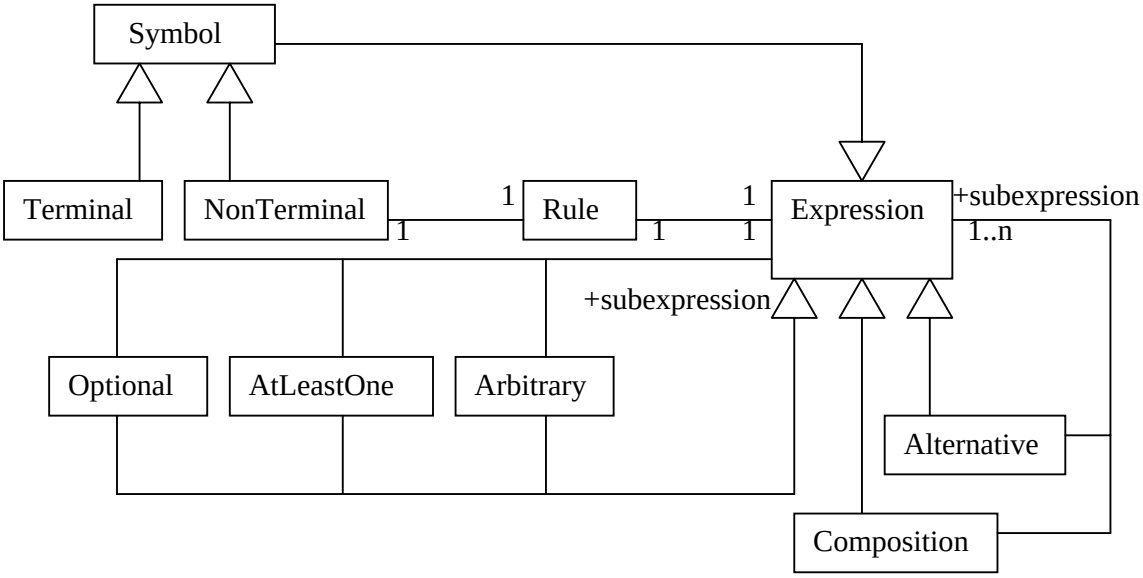

Fig 2.1 EBNF grammar meta-metamodel

EBNF grammars, so they could be regarded as concepts class. The grammar concepts Composition and Alternative seem to be more difficult to handle. We can dig into the semantics of the two concepts. Let's think of a simple example of Composition:

A := B C D E ;

A is the NonTerminal that is to be replaced by the sequence of B C D E. In fact, this reveals the relationship between A and B, C, D and E. See the simple diagram in Fig 2.2, a NonTerminal A is related to a B, a C, a D and an E, with the multiplicity 1 to 1. Obviously, Composition in EBNF can be represented by concept Association in object-oriented models.

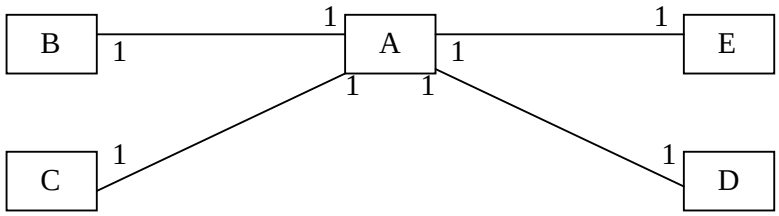

Fig 2.2 Composition of EBNF in object-oriented model

As to concept Alternative, we can also analyze an example:

A := B | C | D | E ;

In this example, NonTerminal A can be replaced exclusively by B, C, D or E, which means, when a replacement occurs, only one of B, C, D and E is chosen to replace A. Then the selected one is regarded as a specialized A, or a variant of A. Therefore, Alternative is mapping to concept Generalization in object-oriented models. And in this example, the generalization is used instead of alternatives as shown in Fig 2.3.

Optional, AtLeastOne and Arbitrary in Fig 2.1 are similar to Composition in semantics, only with the multiplicity different. We can therefore deal with them in the same way as Composition ---- using Association to represent them in object-oriented models.

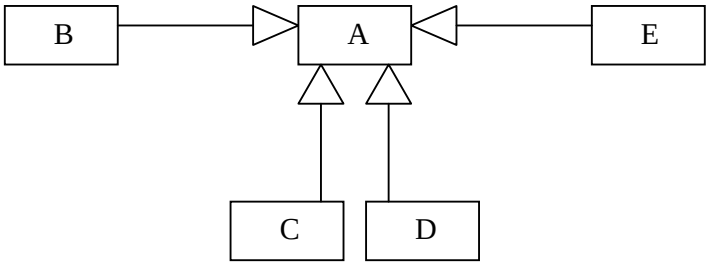

Fig 2.3 Alternative of EBNF in object-oriented model

After analyzing the semantics of the EBNF concepts in Fig 2.1, we have found their representations in object-oriented models. Let's put the representations together to form a more common and object-oriented-closer meta-metamodel, as shown in Fig 2.4. *[4]*

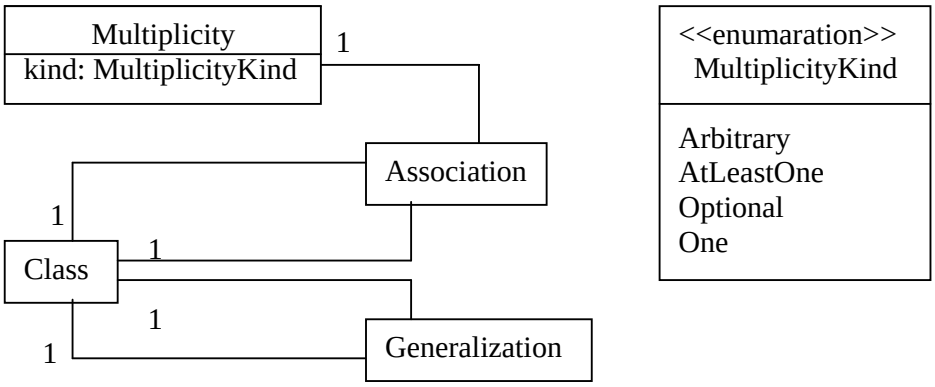

Fig 2.4 A more common and object-oriented-closer meta-metamodel

2.4 Transformation rules

Through the analysis in Section 2.3, we are already ware of the mapping relationships between EBNF concepts and object-oriented concepts, and a more common and object-oriented-closer meta-metamodel has been built. The rules for the transformation from EBNF grammars to metamodel is extracted from the mapping relationships, because what we want to achieve is that EBNF grammars are represented in object-oriented models.

The transformation rules as follows are cited from the paper "A Metamodel for SDL-2000 in the Context of Metamodelling ULF":

1. *Every symbol is represented through a class.*

2. *A rule with a single symbol on the right is represented through an association that associates the class representing the left hand symbol with the class representing the*

right hand symbol.

3. *A rule with a composition on the right is represented through an association for every composed sub-expression.*

4. *A rule with an alternative on the right is represented through a generalization for every alternated sub-expression.*

5. *A sub-expression consisting only of a single symbol is represented through that symbol's class.*

6. *A sub-expression that is a composition or an alternative is represented through a new class, with a so-far unused name. The composition or alternative is transformed as in 2 or 3, but with the new class as the left hand representative.*

7. *A sub-expression of multiplicity kind, that is part of a composition, is transformed to an equivalent multiplicity kind of the proper association end.*

8. *An expression of multiplicity kind or a sub-expression of multiplicity kind, that is part of an alternative, is represented through a new class with a so far unused name. An association is introduced between that new class and the class that represents the multiplied sub-expression, with proper multiplicity.* *[4]*

3 ANTLR and Octopus

In Chapter 2, we have discussed the approach to build EBNF metamodel and the transformation rules were also created. We may find that all the building and transforming activities are based on a precondition that EBNF grammars are parsed and their concept constructs are all recognized. Parsing grammars and recognizing their concept constructs can be considered as compiling. In the procedure of compiling, the codes written in a programming language are taken as input, and all the grammar units are recognized as tokens, then the token stream is parsed for its syntactic structure, finally the semantics of the codes' syntactic structure are analyzed, and after optimizing, the target code is generated. This is a general procedure of compiling, and we already find it too complicated to build a similar compiler ourselves to compile the EBNF grammars and some languages abiding by certain EBNF grammars. Fortunately, there is a ready-to-use tool to help us. In this chapter, this tool, which is named ANTLR, is introduced and illustrated. And another tool to deal with UML files is briefly introduced in the last part of this chapter.

3.1 ANTLR

ANTLR, ANother Tool for Language Recognition, is a tool that accepts grammatical language descriptions and generates programs that recognize sentences in those languages.*[5]* We can consider ANTLR as an automatic quasi-compiler generator. In Fig 3.1 is shown an ANTLR plug-in for Eclipse.

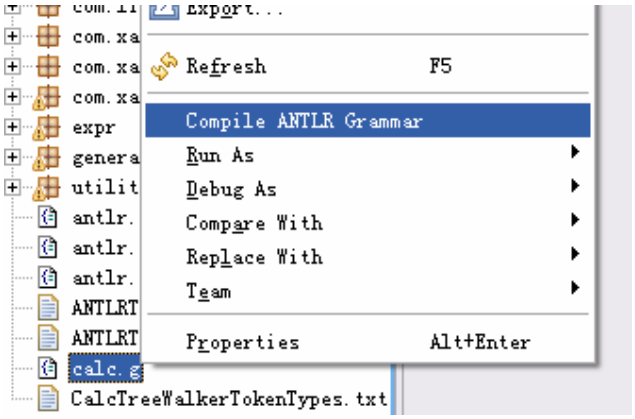

Fig 3.1 ANTLR plug-in for Eclipse

3.1.1 ANTLR Grammar

ANTLR accepts grammatical language descriptions and generates quasi-compilers, so the grammatical language descriptions are the base for the code generation and obviously

important. All the grammatical descriptions are stored as .g files only specified and recognized by ANTLR, see *calc.g* in Fig 3.1.

ANTLR provides a special grammar for the grammatical language descriptions, the .g files. This grammar is based on EBNF, which means a .g file is mainly composed of a grammatical languages description that is using EBNF for the description. Furthermore, as part of a translator, we may augment our grammars with simple operators and actions to tell ANTLR how to build ASTs (Abstract Syntactic Tree) and how to generate output. ASTs are syntactic structures of codes presented in the form of trees, e.g. in Fig 3.2.

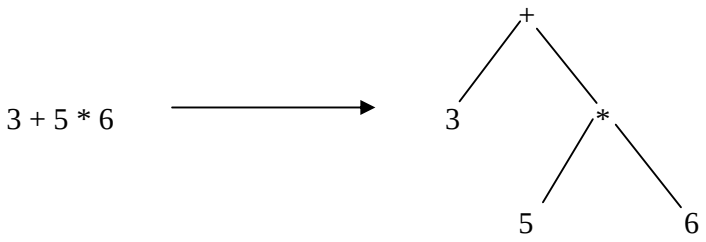

Fig 3.2 AST of Expression 3+5*6

The following is a template of ANTLR grammar, with each structural section commented:

```
header {
  // stuff that is placed at the top of <all> generated files
}

options { options for entire grammar file }

{ optional class preamble - output to generated classfile immediately before
the definition of the class }
class YourLexerClass extends Lexer;
// definition extends from here to next class definition
// (or EOF if no more class defs)
options { YourOptions }
tokens...
lexer rules...
myrule[args] returns [retval]
   options { defaultErrorHandler=false; }
   :  // body of rule...
   ;

{ optional class preamble - output to generated classfile immediately before
the definition of the class }
class YourParserClass extends Parser;
options { YourOptions }
tokens...
parser rules...

{ optional class preamble - output to generated classfile immediately before
the definition of the class }
class YourTreeParserClass extends TreeParser;
options { YourOptions }
tokens...
tree parser rules...
```

```
// arbitrary lexers, parsers and treeparsers may be included [6]
```

Getting familiar with the template above is best done via example. You may still remember that in Section 2.1, an EBNF grammar for simple arithmetic calculation was given, and just as we said that the grammar for .g files is based on EBNF and .g files mainly contain EBNF grammars, we just use the simple arithmetic calculation grammar as the base of our simple *calc.g* file example.

```
class CalcParser extends Parser;
options {
    buildAST = true;   // uses CommonAST by default
}

expr    :   proexpr ((PLUS^ | MINUS^) proexpr)* SEMI!
        ;

proexpr
        :   powpr ((MUL^ | DIV^) powpr)*
        ;

powpr   :   atom (POW^ atom)?
        ;

atom    :   INT
        ;
```

Fig 3.3 Part 1 of calc.g ---- CalcParser

In Fig 3.3, the grammar part was given before, but there is still some tiny differences. First, each arithmetic operator, *PLUS*, *MINUS*, *MUL*, *DIV* and *POW*, is postfixed with a caret sign '^'. This is an ANTLR directive specific to the creation of ASTs, it specifies that the token the caret postfixes should become the root of the current AST or AST subtree, just like '+' and '*' in Fig 3.2. Second, an excalmatory mark is attached behind *SEMI*, which is also an ANTLR directive for ASTs, it specifies that the tokens postfixed with '**!**' will not appear in the resulting ASTs. Those special signs will be illustrated in Section 3.1.3.

Besides the carets and the excalmatory mark, we should pay attention to the part above *expr*. There are two parts, one is the first line, and the other is the following block. The first line declares what role this fragment of codes will act as in the later generated program. In ANTLR, all ANTLR grammars are subclasses of *Lexer*, *Parser*, or *TreeParser*. *Lexer* is taking the stream of codes in the target language as input and generates a corresponding token stream; then *Parser* parses this token stream and generates a syntactic structure, such as AST; *TreeParser* is a tree walker, which traverses the generated AST and does some further work. In this example, we should think over at the syntactic level, therefore a Parser subclass should be built. The first line is declaring which type of subclass should be built in the later generated program.

The second part in Fig 3.3 is a block titled *options*. In this block, some optional attributes can be set, just like **`buildAST = true;`**, which requires *CommonAST* be built by default.

Besides setting some attributes, some our own created variables can be declared here, even user created methods can be defined here.

In Fig 3.4 is shown Lexer definition part of *calc.g* file. That is almost the same as what we have in Chapter 2. Only except for **`{ _ttype = Token.SKIP; }`**, this line of codes is just like a script embedded in the lexer grammatical definition. All script-like codes in .g files are marked with curly brackets, and the scripts would be directly used in the later generated program without being modified, whereas the other grammar-like codes in .g files are automatically translated to programming language codes, such as Java language. **`_ttype`** is a variable that already exists or would be newly declared. **`Token.SKIP`** is a predefined constant in ANTLR, which means that those characters would be ignored by when scanned by the lexer. The other parts of this lexer definition are of no need to explain more.

```
class CalcLexer extends Lexer;

WS      :       (' '
        |       '\t'
        |       '\n'
        |       '\r')
                { _ttype = Token.SKIP; }
        ;

MUL     :   '*'
        ;

DIV     :   '/'
        ;

PLUS    :   '+'
        ;

MINUS   :   '-'
        ;

POW     :   '^'
        ;

SEMI    :   ';'
        ;

INT     :   ('0'..'9')+
        ;
```

Fig 3.4 Part 2 of calc.g ---- CalcLexer

In this example, we don't define a *TreeParser*, it is optional. All the three types of classes are optional and their relative sequence in a .g file is not fixed. .g files could be far more complex, but here for this arithmetic grammar, a simple *calc.g* is enough for ANLTR to generate a useable simple arithmetic calculation compiler. If you are interested in ANTLR grammar or you need more details, further materials at **www.antlr.org** are waiting for you.

3.1.2 Compiler Generation by ANTLR

As we have talked, ANTLR accepts grammatical language descriptions, which are written in .g files and based on EBNF grammars, and generates programs from .g files accordingly. The generated programs are actually compilers for the codes in programming languages that are grammatically described in the .g files.

Last section we got to know how a .g file is built, and a sample .g file *calc.g* was given, which was using the simple arithmetic calculation's EBNF grammar that was shown in Chapter 2. So we already have a .g file now, and what we should do next is straightforward, to generate the compiler for simple arithmetic calculation from *calc.g* file.

In Fig 3.1, a popup menu with a menu selection "Compile ANTLR Grammar" was shown, this menu selection is from the integrated ANTLR plug-in for Eclipse. Obviously, clicking this selection will make ANTLR execute ---- compiling .g files and generating a compiler automatically.

Let's try this operation on *calc.g* as what was displayed in Fig 3.1. Undoubtedly, a compiler consisting of several Java files is generated, see Fig 3.5.

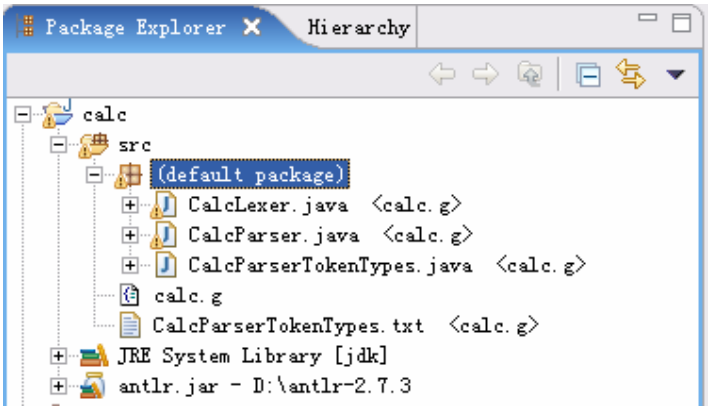

Fig 3.5 Generated compiler files

By default, all resulting Java files are put in default package, which means the Java classes defined in those files don't have package declarations. There are three resulting Java files, *CalcLexer.java* that is representing a lexer corresponding to the lexer part in *calc.g* file, *CalcParser.java* that is representing a parser corresponding to the parser part in *calc.g* file, and *CalcParserTokenTypes.java* that defines all token types recognized by the lexer. Another *CalcParserTokenTypes.txt* also records all the token types.

One more point to mention is that we have to include the library *antlr.jar* which is contained in the ANTLR tool package you can download from **www.antlr.org**. Otherwise, some codes in the resulting Java files could not be resolved, because some classes in ANTLR are used in the codes.

3.1.3 Build a Client of the Compiler

Now, we have a compiler to parse simple arithmetic calculations. But how can the compiler be started? Therefore, we still need a client to build lexer and parser objects, then let them accept simple arithmetic calculations as input and build resulting ASTs as output.

Here we build a client to run the compiler. The codes are as follows:

```
import antlr.CommonAST;
import antlr.debug.misc.ASTFrame;
import java.io.DataInputStream;

public class CalcTest {

    public static void main(String[] args) {
        try {
              DataInputStream input = new DataInputStream(System.in);
              CalcLexer lexer = new CalcLexer(input);

              CalcParser parser = new CalcParser(lexer);
              parser.expr();

              CommonAST parseTree = (CommonAST)parser.getAST();
              String result = parseTree.toStringList();
              System.out.println(result);
              ASTFrame frame = new ASTFrame("The tree", parseTree);
              frame.setVisible(true);

            } catch(Exception e) { System.err.println("Exception: "+e); }
    }
}
```

Fig 3.6 CalcTest.java ---- Client for Calc Compiler

In this client, the keyboard input in the console is taken from the system input stream, and passed to build a *CalcLexer* instance. Generally, a *CalcLexer* instance requires input in the form of an input stream or an array of characters.

An *CalcParser* instance is built with a *CalcLexer* object as an argument. Here this *CalcLexer* object represents a token stream. The *CalcParser* instance's *expr()* method is called to start parsing. It extracts tokens from the token stream, analyzes their syntactic structure and then built an AST which is returned by calling *getAST()* method. The resulting AST can be output to the console by method *toStringList()* of class *CommonAST*, which is defined in ANTLR to represent an AST. ANTLR also provides a class *ASTFrame*, which is a subclass of *javax.swing.JFrame*, to visualize ASTs.
Let's run the client and type in the console an arithmetic expression "**3+5*6^9;**" as input.

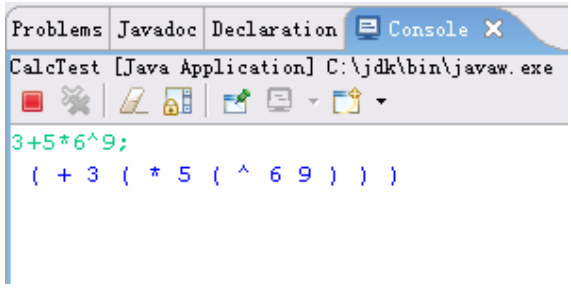

Fig 3.7 Console output for arithmetic expression "**3+5*6^9;**"

The resulting console output is displayed as Fig 3.7. The output String **(+ 3 (* 5 (^ 6 9)))** is generated by the method *toStringList()* of class *CommonAST*. This string shows in text the syntactic structure of the input arithmetic expression. **6^9** is a power expression and built as a subtree first; then **6^9** is thought of as a single unit that is multiplied by 5, so that a product expression **5*(6^9)** is built as a subtree; finally, the product expression is thought of as a single unit and added with 3, therefore a top tree is built, which is representing **3+(5*(6^9))**.

Don't forget we still have an ASTFrame instance to show the resulting AST, as shown in the following picture:

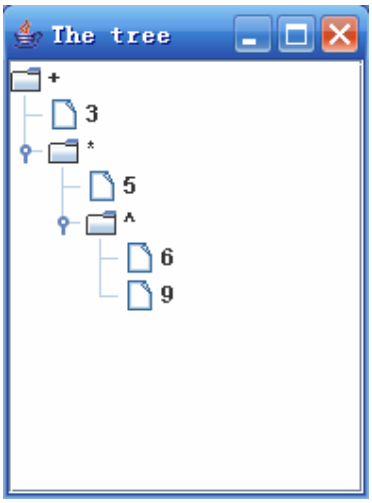

Fig 3.8 Visualized AST of arithmetic expression "**3+5*6^9;**"

Given in Section 3.1.1 that the token postfixed with a caret '^' would be set as the root of the resulting AST or one of its sub ASTs, and the token postfixed with an excalmatory mark '**!**' would not appear in the resulting AST, we might find in this visualized AST that '+', '*****' and '^', each of which is postfixed with a caret, all appears as the roots of ASTs, whereas the last sign '**;**', which is postfixed with an excalmatory mark, does not appear in this AST.
This visualized AST tells us that AST is a very straightforward syntactic structure for the compiled codes. Thus, if we want to do some syntax related work, this tree structure would be very helpful, because we can add codes that will be executed when the tree structure is being traversed. In the next chapter, you will see how this tree structure is utilized.

3.2 Octopus

Octopus, which stands for "OCL Tool for Precise UML Specifications", is a tool to support the use of OCL. *[7]* Fig 3.9 and Fig 3.10 show the Octopus plug-in for Eclipse.

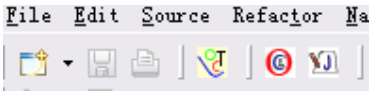

Fig 3.9 Octopus plug-in for Eclipse --- in Menu Bar

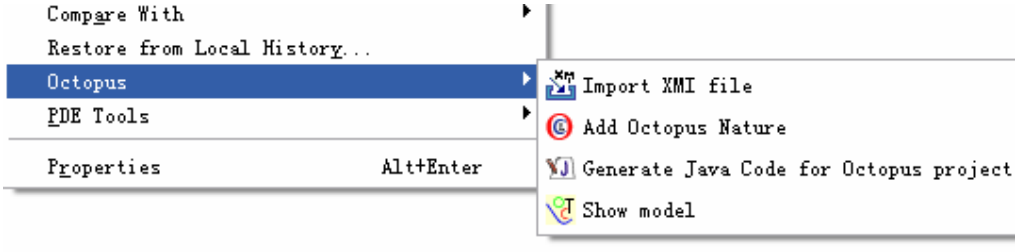

Fig 3.10 Octopus plug-in for Eclipse --- in Popup Menu

We do not involve OCL in this project, so the support to OCL seems useless to us. However, Octopus has another feature that it is able to transform the UML model, including the OCL expressions, into Java codes.

3.2.1 .uml files as UML model

In Octopus, UML models are represented by .uml files. .uml files are text representation of UML diagrams. This text-based UML model will be used as our metamodel of EBNF grammars. Here we will see how an .uml file is by an example.

You may still remember we have built EBNF grammar meta-metamodel in Fig 2.1, which is a UML class diagram. It's a good sample UML model for us to represent in an .uml file. But considering the actual use of this meta-metamodel in this project, we need to modify it a little. When we try to transform the EBNF grammars to object-oriented model, according to the transformation rules given in Section 2.4, an EBNF rule is represented as an association between a NonTerminal and a combination of other EBNF concept constructs. For example, **powpr : atom (POW^ atom)? ;** is a rule, in a UML class diagram, it would be represented by a class **powpr**, another class which stands for the composition of **atom** and **(POW^ atom)?**, and a one-to-one association between the two classes. Obviously, we don't use a class **rule** when we build this UML class diagram for an actual EBNF grammar. Therefore, I decide to remove this rule from Fig 2.1. The resulting UML class diagram for EBNF grammar meta-metamodel is as follows:

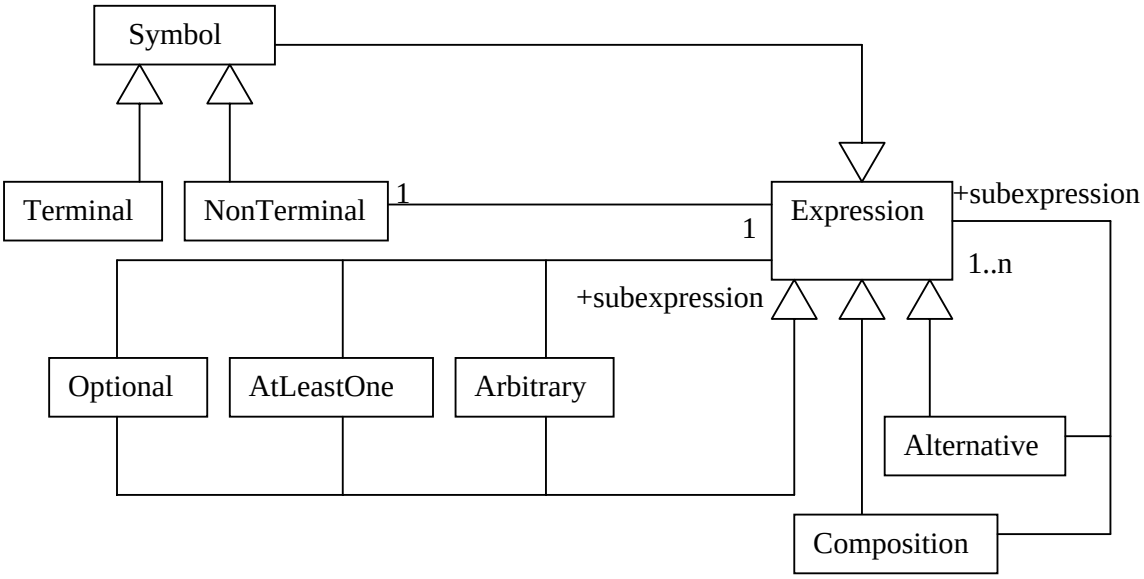

Fig 3.11 Modified EBNF grammar meta-metamodel

Given this modified EBNF grammar meta-metamodel, we can translate it into an .uml file. This *BNF.uml* could be created like follows:

```
<package> bnfmetamodel

<class> Expression
<endclass>

<class> Symbol <specializes> Expression
<endclass>

<class> Terminal <specializes> Symbol
<endclass>

<class> NonTerminal <specializes> Symbol
<endclass>

<class> Alternative <specializes> Expression
<endclass>

<class> Composition <specializes> Expression
<endclass>

<class> Optional <specializes> Expression
<endclass>

<class> AtLeastOne <specializes> Expression
<endclass>

<class> Arbitrary <specializes> Expression
<endclass>

<associations>
+NonTerminal.<noName>[1] <-> +Expression.<noName>[1];
+Expression.subexpression[1] <-> +Optional.<noName>[0..*];
+Expression.subexpression[1..*] <-> +AtLeastOne.<noName>[0..*];
```

```
+Expression.subexpression[1..*] <-> +Arbitrary.<noName>[0..*];
+Expression.subexpression[1..*] <->
                +Composition.<noName><ordered><composite>;
+Expression.subexpression[1..*] <-> +Alternative.<noName>[0..*];

<endpackage>
```

In this *BNF.uml*, we use `<class>` and `<endclass>` tags to declare classes drawn in the UML class diagram, and `<specializes>` tag is used to indicate that the declared class inherits from the class specified after this tag. In `<associations>` section, '+' is the *public* modifier; the digits in square brackets '[]' are the multiplicities of associations; `<noName>` is the default role of an participant in an association; `<ordered><composite>` indicates that a *Composition* should be a composite of *Expression*s which act as subexpressions and should be ordered, to be more detailed, when Octopus automatically generates Java codes according to this .uml file, this `<ordered>` tag will require that class Composition have a *List* instance, instead of a *Set* instance, to hold all its subexpressions in order to keep some certain sequence; finally, `<package>` and `<endpackage>` tags are used to specify the package name which would be declared in each classes. *[7]*

3.2.2 Automatic Java codes generation from .uml files

After creating an .uml file, we can start generating Java codes from it. That's very simple. First, the Java project should be added with Octopus Nature, click the popup menu selection "Add Octopus Nature" in Fig 3.10, so that two folders "expressions" and "model" are created, "expressions" folder is for OCL, and "model" folder contains all .uml files; then we put *BNF.uml* in a new folder "bnfmetamodel" corresponding to the package name specified in this .uml file, and "bnfmetamodel" should be contained in "model" folder, see Fig 3.12;

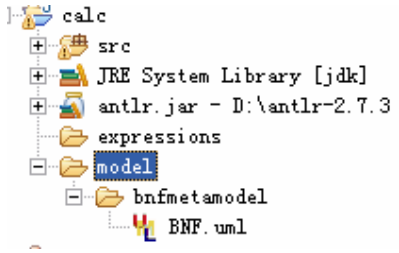

Fig 3.12 *BNF.uml* in "model" folder

finally, just click the popup menu selection "Generate Java Code for Octopus project" as shown in Fig 3.10. Then the Java code is generated as expected:

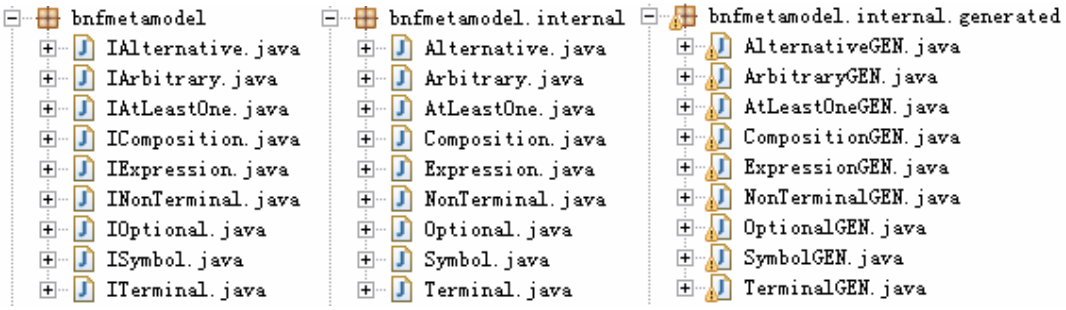

Fig 3.13 Generated Java files from *BNF.uml*

In Fig 3.13, the generated Java code has three packages, package *bnfmetamodel* provides the interfaces of the classes defined in *BNF.uml*; the classes in package *bnfmetamodel.internal. generated* implement those interfaces, and in general, we do not directly create objects of the classes in this package; instead, we define their subclasses in *bnfmetamodel.internal* package, which we can instantiate.

4 Building Prototype

4.1 A Short Review

After the explanation of building metamodel in Chapter 2 and the introduction of ANTLR and Octopus in Chapter 3, we should make a short review of what we have done.

In Chapter 2, EBNF is first introduced with a sample given, which is the grammar for simple arithmetic calculation expressions. Then we compared and discussed two main methodologies for describing syntax definition, metamodel and context free grammars. After we got the point that metamodel is more suitable in this project, we built an EBNF grammar meta-metamodel, and modified it to an object-oriented-closer version, in which the mapping between EBNF meta-concepts and object-oriented concepts were represented, and we therefore summarized transformation rules from this mapping.

In Chapter 3, ANTLR appeared with a sample ANTLR grammar, *calc.g*, which was built on base of the sample EBNF grammar for simple arithmetic calculation expressions. Certainly, we did not forget explaining every detail of *calc.g*. Furthermore, we used ANTLR to compile *calc.g* and got some Java files that formed a simple compiler for simple arithmetic calculation expressions, and then we built a client program to run this simple compiler. Testing on an arithmetic expression, its resulting AST was exhibited. Another part of Chapter 3 is about Octopus, where we modified a little the EBNF meta-metamodel (a UML class diagram) and converted that class diagram to *BNF.uml* that could be recognized by Octopus. Here an .uml represented an object-oriented metamodel. Finally, we used Octopus to generate Java classes declared in *BNF.uml*.

We can list all artifacts we finished and mentioned above as follows:

1. EBNF grammar for simple arithmetic calculation expressions;
2. EBNF grammar meta-meta model;
3. Transformation rules for converting common EBNF concepts to object-oriented metamodels;
4. *calc.g* based on 1;
5. Generated compiler from 4 for simple arithmetic calculation expressions;
6. *BNF.uml* based on 2;
7. Generated classes from 6 corresponding to the classes in 2.

4.2 Move to Next Steps

In order to build a prototype, we need to keep the work not complex. A good point we can catch to make it simple is the grammar to be handled in the prototype. In fact, on the way leading to here, we already considered this matter. You must have found that we chose a

simple instance EBNF grammar that is the EBNF grammar for simple arithmetic calculation expressions.

With this simple grammar chosen, we should think over our tasks again. In the prototype we are developing, syntax check functionality for simple arithmetic calculation expressions should be provided behind a console as an Eclipse plug-in, where we can type in simple arithmetic calculation expressions and obtain the syntax analysis information as result. Moreover, maybe we would better make other syntax related operations easier.

Syntax check and other syntax related operations involve EBNF grammars handling. In stead of directly dealing with context free grammars, we proposed to use metamodel. Transformation from EBNF grammars to metamodels is the key point. Through two chapters' discussion, we know .g files could be thought of as EBNF grammars, whereas .uml files could be thought of as object-oriented metamodels. Obviously, transforming .g files to .uml files is just transforming EBNF grammars to metamodels.

When the transformation functionality is available, given any .g files, we could get the corresponding .uml files. Of course, here we only want the .uml file for *calc.g*. And then we could use Octopus to generate Java files, each of which should define a Java class and stand for a grammar concept of EBNF grammar for simple arithmetic calculation expressions that we have mentioned many times. In addition, the objects of each class should store some important information, such as what grammar concept the objects of this class stand for and some other instance information, e.g. an integer has value 5.

Meanwhile, we could use ANTLR to compile *calc.g* and get a generated compiler. With this compiler, any input simple arithmetic calculation expressions could be parsed and its AST grammar tree could be built accordingly. At this moment, we should have the AST of the input simple arithmetic expression, and Java classes for arithmetic grammar concepts in *calc.g*. Therefore, while traversing the AST, we could instantiate objects of the Java classes for any encountered grammatical tree nodes. Because each Java objects should get in traversing and store some important grammatical and instance information, in the console plug-in, we could easily get the syntax information of the input simple arithmetic calculation expressions.

Until to now, I think the entire procedure of what we planned to do is clear and not intangible any more. And it would be better if we could draw a big picture of the entire procedure. Let's have a look at Fig 4.1 in the next page.

In Fig 4.1, a clear working procedure of this prototype to be built is appearing in front of our eyes. We can find in this figure some artifacts we have finished and listed in Section 4.1. And also, we may notice the other parts we have not yet, such as Transformer, calc AST Traverser, and other functional units.

From the next section on, we will go along the procedure depicted in Fig 4.1, and implement each unfinished part, finally build up the target prototype.

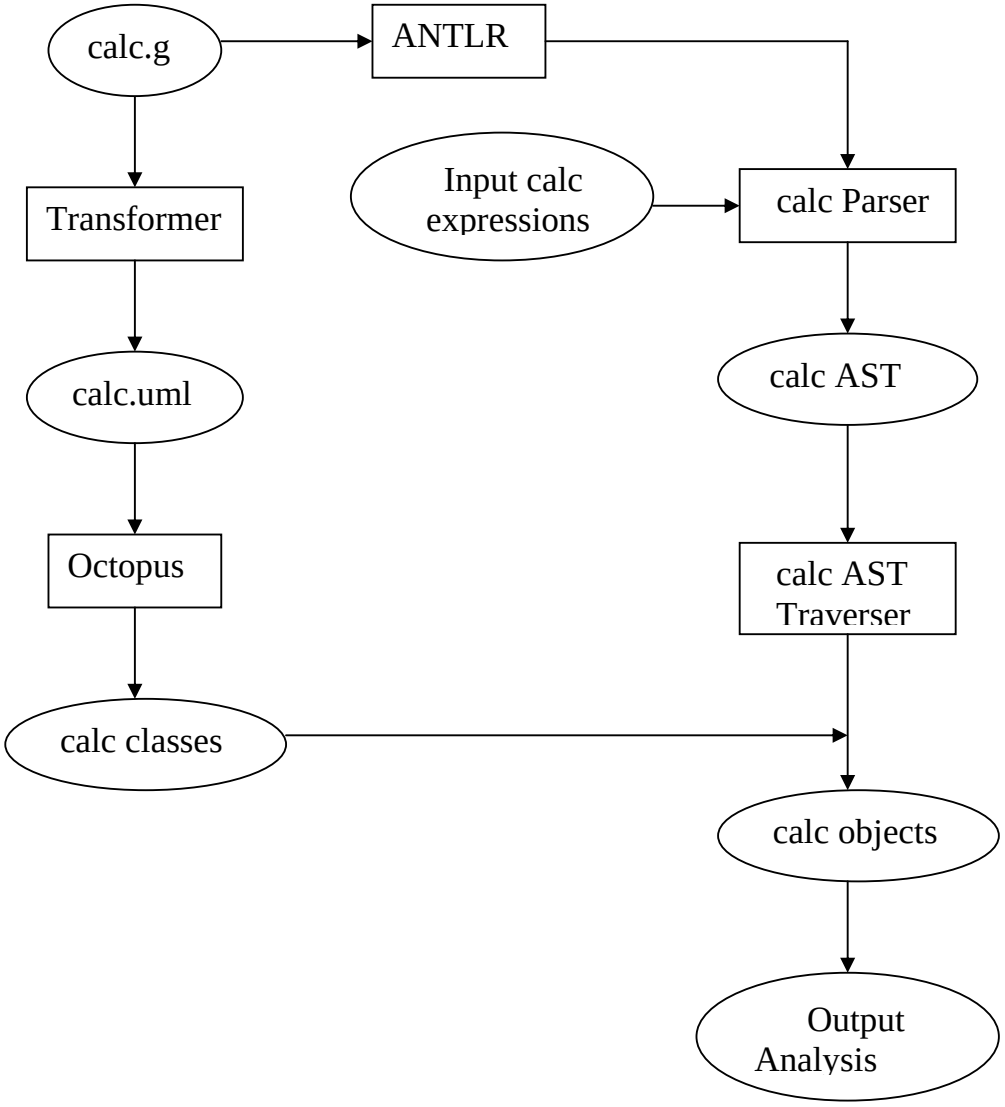

Fig 4.1 Workflow of the Prototype

4.3 Implementing Transformer

The transformer is used to as a tool that takes .g files as input and generates .uml files as output. In our imagination, in order to process a .g file, this .g file should be first parsed and its AST should be built, then during traversing its AST, we can generate UML concept objects corresponding to the nodes of its AST. But in fact, it was not so easy as what we just thought. It took a lot of time and effort. Let's now go along the way once I went along.

4.3.1 First Try for Transformer

How could we parse a .g file? Just think about the way we parse a simple arithmetic calculation expression: we first created *calc.g* that was based on the EBNF grammar for

simple arithmetic calculation expressions, then we used ANTLR to compile *calc.g* and obtained a parser consisting of several Java files, and this parser is to some extent a compiler for simple arithmetic calculation expressions.

Now, we want parse a .g file, therefore, we can go in the same way to build a parser for all .g files. However, first of all, we need the EBNF grammar for all .g files that we can create an ANTLR.g or something like that which is based on the .g files' EBNF grammar. It seems not easy to summarize the EBNF grammar. Fortunately, the developer of ANTLR already provided such an ANTLR-meta-grammar which consists of three .g files and some Java files as follows:

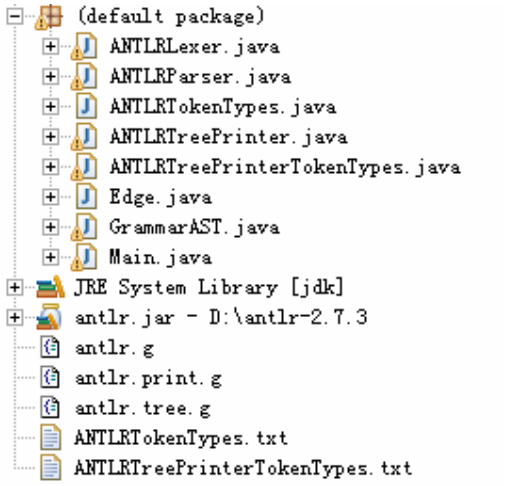

Fig 4.2 ANTLR-meta-grammar

The three .g files seem very simple, but actually they contain the complex grammar of all .g files. *antlr.g* consists of a lexer and a parser for .g files, whereas *antlr.print.g* and *antlr.tree.g* are treeparsers, and they are only a little different, so we only need one treeparser from either of them, here is *ANTLRTreePrinter.java*. The first five Java files were generated from the three .g files. *Edge.java* and *GrammarAST.java* were provided by the developer.

This ANTLR-meta-grammar can parse any .g file. Do you remember we have generated some Java classes standing for EBNF meta-concepts at the end of Chapter 3? Based on it, we need to instantiate the objects from those classes for each node in the AST of a parsed .g file.

However, the implementation of instantiating objects for the AST tree nodes is not so straightforward. I tried with two different means, the effective method will be given in the next section, and here I first document the failed one, which would help you get to understanding the effective method.

At the beginning of the implementation, the existing treeparser, *ANTLRTreePrinter.java*, easily came into my eyes, obviously, if I could add some code to it, it would instantiate the objects we want. But the Java code of this treeparser is somehow complex, and the following code fragment is taken from it:

ANTLRTreePrinter.java

```
public final void rule(AST _t) throws RecognitionException {
    AST rule_AST_in = (AST)_t;
    AST r = null;
    try {      // for error handling
        AST __t13 = _t;
        AST tmp5_AST_in = (AST)_t;
        match(_t,RULE);
        _t = _t.getFirstChild();
        r = (AST)_t;
        match(_t,RULE_REF);
        _t = _t.getNextSibling();
        out(r+" : ");
        AST __t14 = _t;
        AST tmp6_AST_in = (AST)_t;
        match(_t,BLOCK);
        _t = _t.getFirstChild();
        block(_t);
        _t = _retTree;
        _t = __t14;
        _t = _t.getNextSibling();
        AST tmp7_AST_in = (AST)_t;
        match(_t,EOR);
        _t = _t.getNextSibling();
        out(";\n");
        _t = __t13;
        _t = _t.getNextSibling();
    } catch (RecognitionException ex) {
        reportError(ex);
        if (_t!=null) {_t = _t.getNextSibling();}
    }
    _retTree = _t;
}
```

Fig 4.3 Code fragment from *ANTLRTreePrinter.java*

Manually adding code to such a Java file is error-prone. Don't forget this Java file was generated from *antlr.print.g*, as we have known in Chapter 3, Java code could be inserted into .g files as scripts. Moreover, the main grammar part of *antlr.print.g* is not so complex as shown in Fig 4.4. Therefore, we can add Java code to *antlr.print.g*, and then use ANTLR to compile this .g file and get a new *ANTLRTreePrinter.java*. And the Procedure of instantiating EBNF meta-concept objects is accordingly drawn as Fig 4.5.

Unfortunately, maybe because the added code is incompatible with the automatically generated code, or because it's somehow error-prone to handle the recursive grammar constructs, I had to make a lot of structural changes on *antlr.print.g*. Even though the new treeparser generated from the modified *antlr.print.g* worked well on *calc.g*, but to complex .g files, it might not work, I had to drop this method.

```
grammar
    :   ( classDef ) // ignore 2nd (lexer) grammar for now
    ;

classDef
    :   #( PARSER rules )
    |   #( TREE_PARSER rules )
    |   #( LEXER rules )
    ;

rules
    :   ( rule )+
    ;

rule
    :   #( RULE r:RULE_REF {out(r+" : ");} #(BLOCK block) EOR {out(";\n");} )
    ;

block
    :   {out(" (");} alternative ( {out(" | ");} alternative)* {out(")");}
    ;

alternative
    :   #( ALT (element)+ )
    ;

element
    :   atom
    |   #(NOT {out("~");} element)
    |   #(RANGE atom {out("..");} atom)
    |   #(CHAR_RANGE CHAR_LITERAL {out("..");} CHAR_LITERAL)
    |   ebnf
    |   tree
    |   #( SYNPRED block ) {out("=>");}
    |   ACTION
    |   SEMPRED
    |   EPSILON {out(" epsilon ");}
    ;

ebnf:   #( BLOCK block ) {out(" ");}
    |   #( OPTIONAL block ) {out("? ");}
    |   #( CLOSURE block )  {out("* ");}
    |   #( POSITIVE_CLOSURE block ) {out("+ ");}
    ;

tree:   #(TREE_BEGIN {out(" #(");} atom (element)* {out(") ");} )
    ;

Atom    {out(" "+#atom.toString()+" ");}
    :   RULE_REF
    |   TOKEN_REF
    |   CHAR_LITERAL
    |   STRING_LITERAL
    |   WILD_CARD
    ;
```

Fig 4.4 The main grammar part of *antlr.print.g*

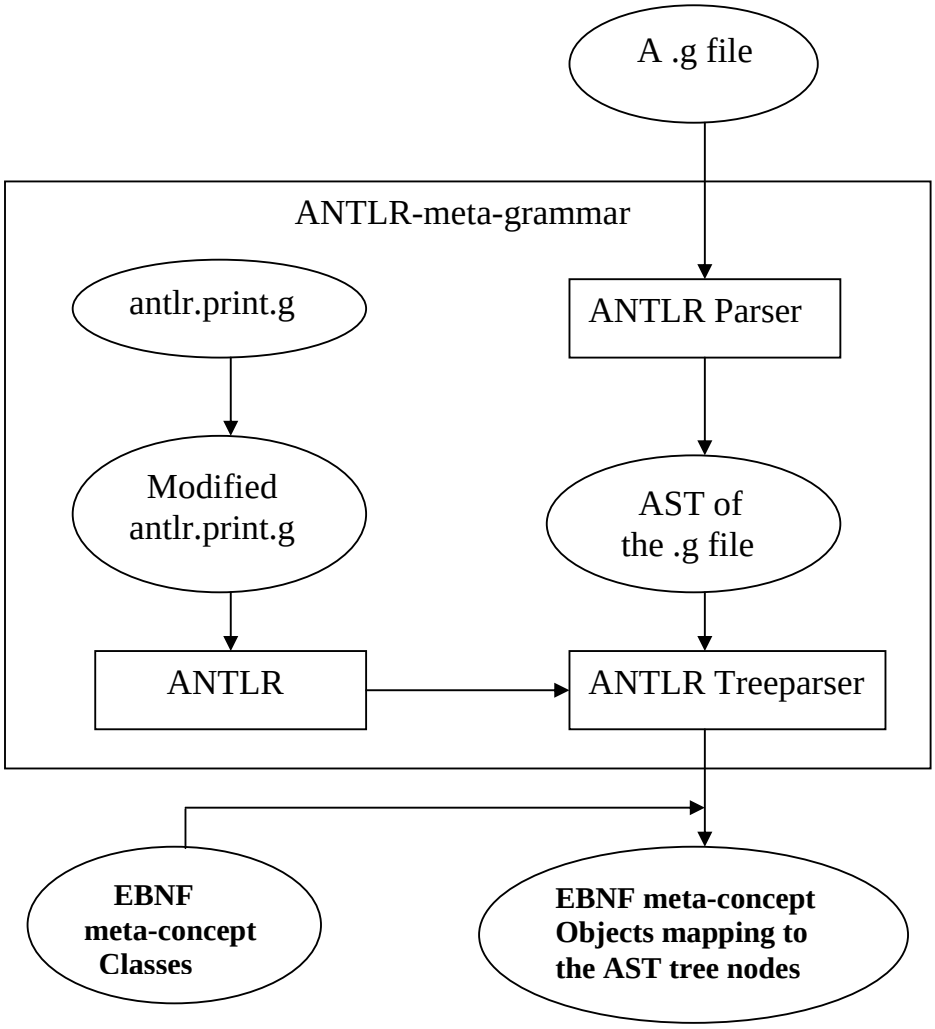

Fig 4.5 Procedure of instantiating EBNF meta-concept Objects at first try

4.3.2 Implementing Traverser in another way

As we talked in the last section, either easily adding code to *ANTLRTreePrinter.java* or modifying *antlr.print.g* to generate a new treeparser is error-prone, problematic and not suitable.

If we could not use the existing treeparser, should we create our own treeparser? Yes, why not? But it seems hard to start off. Don't worry and you may remember the code fragment of *ANTLRTreePrinter.java* in Fig 4.3, which is a method ***rule (AST _t)***. It has an argument that is an ***AST*** object reference. In Section 3.1.3, we have known that a ***CommonAST*** object is representing an AST. After checking *anltr.jar*, we should find that class ***CommonAST*** is a subclass of ***BaseAST***, whose superclass is ***AST***. So an ***AST*** object is undoubtedly representing an AST. What an AST is has been given in Section 3.1.1, here we still need a previous figure to show ASTs and how to traverse an AST.

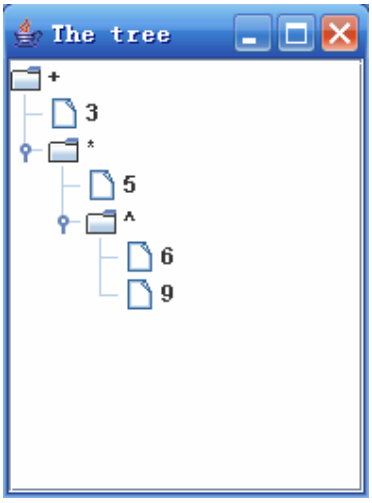

Fig 4.6 AST of arithmetic expression "**3+5*6^9;**" from Fig 3.8

An AST is composed of two ASTs (or more in other cases), each of which could be a simple tree leaf or a complex subtree, e.g. "+" is the root of the top AST, it has two ASTs as children, one is a tree leaf "3", and the other is a subtree whose root is "*". Thus traversing an AST is straightforward: when you get an AST, the root is available to be processed, you can determine which type the AST is or which operator the root is; after processing the root, just move to process the first child and then process the second child, which is the next sibling of the first child of the root. All the tree traversing actions are provided by class ***AST***. We can find those actions in Fig 4.3, such as ***getFirstChild()***, ***getNextSibling()***.

Therefore, we get to understanding the main workflow of the code fragment in Fig 4.3. The method ***rule (AST _t)*** takes an ***AST*** object reference as an argument, in the method body, this AST is traversed. During the traversing, each tree leaf should be syntactically matched, whereas each subtree should be determined for its type and delegated to another method to process, e.g. when a *block* construct is determined, the processing of it is delegated to another method ***block (AST _t)***, which is very similar to ***rule (AST _t)*** and only processes ***block*** constructs.

With the code fragment in Fig 4.3 as an example, it is not so difficult for us to create a new treeparser. In order to build a treeparser, we must to get familiar with the AST syntactic structure for .g files. The picture in Fig 4.7 is the visualized AST of *calc.g*. This AST picture clearly illustrates the syntactic structure of all .g files, and it also matches the main grammar part of *antlr.print.g* in Fig 4.4.

I created a class ***GrammarASTTraverser*** as our new treeparser, and we can call it a traverser. Besides its constructor, it only has one method ***traverseGrammarAST(GrammarAST ast)***. In Fig 4.8, there is a little part of this method. Similarly, it takes an AST object reference as the argument, because class ***GrammarAST*** is provided by ANTLR-meta-grammar and inherits from class ***BaseAST***. The type of an AST is the type of its root, so when an AST is passed in, the type of its root should be first determined, here I used a group of if-else blocks, in which the processing to the certain type of AST is provided. The tree node *AST ROOT* was added manually to the AST that was generated by ANTLR but not original. When we use the traverser, only the original ASTs would be passed in. Furthermore, we are only concerned about the syntactic structure of .g files, but not concerned about what the tokens are

composed of, therefore, in this method we only process the AST *parser* and leave the AST lexer.

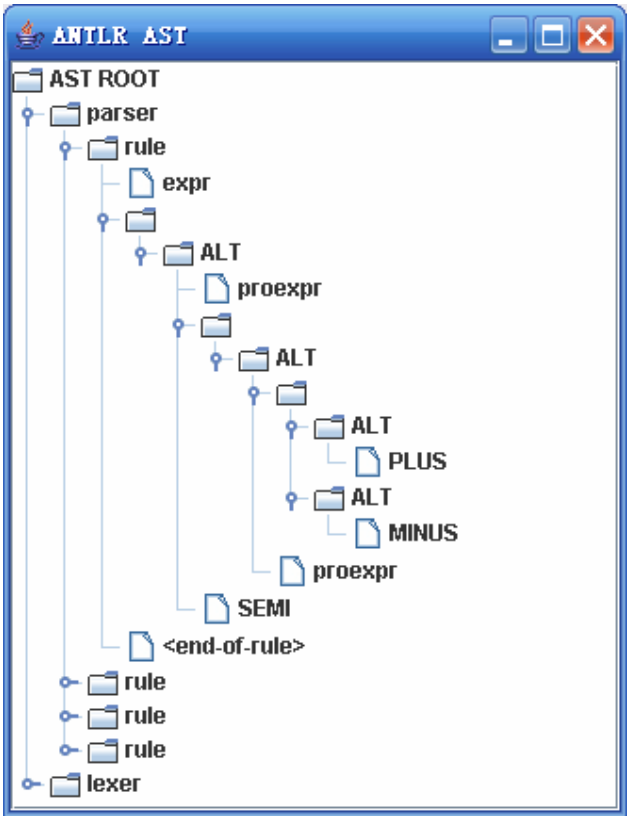

Fig 4.7 Visualized AST of *calc.g*

GrammarASTTraverser.java

```
public GrammarAST traverseGrammarAST(GrammarAST ast) throws Exception{
    GrammarAST tempAst = ast;
    if(tempAst.getType() == PARSER){
        tempAst = (GrammarAST)tempAst.getFirstChild();
        if(tempAst == null) throw new Exception("No construct");
        else {
            while(tempAst != null && tempAst.getType() == RULE){
                tempAst = traverseGrammarAST(tempAst);
            }
        }
    }
    else if(tempAst.getType() == RULE){
```

Fig 4.8 Part 1 of method *traverseGrammarAST(GrammarAST ast)*

Thus AST parser should appear first when starting the traversing. Class ***AST*** provides a method ***getType()*** which can return the type of an AST's root. Through it, AST's type is determined. Variable tempAst is acting as a cursor to point out the current processed AST.

If the argument AST is of type *parser*, the cursor is moved to its first child. From Fig 4.4 and 4.7, we know that the children of an AST of *parser* type should be of type *rule*. So its first child need be determined to be of *rule* type. Then there is a cycle block, but there seems no iteration mechanism. In fact, the recursive method call itself is the iteration. Note that the method **traverseGrammarAST(GrammarAST ast)** has a return value that is a class GrammarAST object reference. Where does this returned AST come from? I did not show the entire method here, so now I reveal that the returned AST is the next sibling of the current processed AST. Then it is not confusing any more. An AST of *parser* type might have many children of *rule* type. To process the first one, we call the recursive method and get the second child as return value. In this way, all children can be traversed within the cycle block.

```
else if(tempAst.getType() == RULE){
    tempAst = (GrammarAST)tempAst.getFirstChild();
    if(tempAst == null) throw new Exception("No construct");
    else{
        if(tempAst.getType() == RULE_REF){
            NonTerminal ntrml = null;
            List ntrmlList = NonTerminal.allInstances();
            for(int i=0; i<ntrmlList.size(); i++){
                String name = ((NonTerminal)ntrmlList.get(i)).getName();
                if(name.equals(tempAst.getText())) {
                    ntrml = (NonTerminal)ntrmlList.get(i);
                    break;
                }
            }
            if(ntrml == null) ntrml = NonTerminal.newNonTerminal(tempAst.getText());
            tempAst = (GrammarAST)tempAst.getNextSibling();
            if(tempAst == null) throw new Exception("No construct");
            else{
                if(tempAst.getType() == BLOCK){
                    Composition mainBlock = Composition.newComposition(ntrml.getName()+"_main");
                    mainBlock.setNonTerminal(ntrml);
                    blockStack.push(mainBlock);
                    tempAst = traverseGrammarAST(tempAst);
                    blockStack.pop();
                }
                else throw new Exception("Unexpected construct");
                if(tempAst.getType() != EOR) throw new Exception("Unexpected construct");
            }
        }
        else throw new Exception("Unexpected construct");
    }
}
```

Fig 4.9 Part 2 of method *traverseGrammarAST(GrammarAST ast)*

Fig 4.9 above shows another part of the method we are talking about. An AST of type *rule* has three children, and the first is a rule reference AST that is mapping to a NonTerminal symbol, because it would be replaced by the right side of a rule. At this moment, an object of class ***NonTerminal*** should be created, but before we do this, we have to check whether such a NonTerminal standing for this rule reference has already been created, and duplication must be avoided. Why duplication is possible? Because in Fig 4.4, *RULE_REF* would be processed as a kind of *Atom*. How could we check duplications? Class ***NonTerminal*** was generated by Octopus, and Octopus equipped each class in package ***bnfmetamodel.internal.generated*** with a static ***ArrayList*** variable, which holds all instances of the class. We only need use a static variable ***allInstances*** of anyone of the classes to get those instances. In Fig 4.9, we get all ***NonTerminal*** objects and use a *for* cycle block to check whether the rule reference NonTerminal is existing. If this NonTerminal is not existing, a new ***NonTerminal*** object standing for the current rule reference is created with the tree node text as its name. I added a static method for creating new instances to each class that Octopus generated, e.g. ***newNonTerminal(String name)***. In this method, the constructor of the class is called and the argument name is set as the name of the new instance.

After processing the rule reference, the second child of the rule is coming, which must be a block, according to Fig 4.4 and Fig 4.7. Here I must explain two important points.

First, Octopus generated classes such as ***NonTerminal*** from *BNF.uml*, and this .uml file was base on EBNF grammar meta-metamodel in Fig 2.1. Thus all the associations depicted in the figure should be implemented in all generated classes, e.g. ***setNonTerminal(NonTerminal ntrml)*** of class ***Composition***. After processing the rule reference NonTerminal and creating a ***Composition*** object as the block, it is important to set up the association between the two components of a rule, where the transformation rules given in Chapter 2 applies and it also complies with the EBNF grammar meta-metamodel in Fig 2.1. Transformation rules are embodied not only by creating correct and suitable EBNF meta-concept objects, but also by setting up exact associations between the objects.

Second, you might notice that the new ***Composition*** object as a *block* of the rule is pushed into a stack before the AST of *block* is processed by calling the recursive method, and it is popped up from the stack after processing. Recursion is very common in programming languages and grammatical descriptions of programming languages, e.g. in Fig 4.4, you may find that a *block* consists of *alternative*s, and an alternative consists of *element*s, then an *element* could be an *ebnf*, which is definitely a *block* from four types. Recursion makes the parsing of grammars more complex. In general, using a stack to hold all nested layers is an effective method and widely adopted in compiling technologies. *[8]* In this way, a nested layer can use the information of its nesting layer that is its parent. Therefore, a ***Stack*** object *blockStack* is created. Every time a *block* is to be processed, the ***Composition*** object as the block is pushed into *blockStack*, and after processing, it is popped up. With *blockStack* used, the nested layer can easily obtain some necessary information of its nesting layer, such as the name information, because the name of an EBNF meta-concept object is got from extending its nesting layer object's name and thus should reveal the layering status information.

After processing the *block*, an EOR is expected, which can be thought of as the third child of the rule. EOR is the abbreviation of End Of Rule, which we can find in Fig 4.7. Reaching this type of AST means the current rule is terminated, and the processing could turn to the next one. All the code of class ***GrammarASTTraverser*** will be attached in Appendix I and is not much more difficult to understand than the already explained two code fragments.

4.3.3 Generating .uml File

After traversing the AST of a .g file, we have the EBNF meta-concepts objects mapping to each tree node in the AST. All these objects are stored in a static ***ArrayList*** instance of each EBNF meta-concept class. In this section, we are using the information held in these objects to build an .uml file.

In order to extract the information held in the objects, we have to traverse all these objects. Similarly, I defined another class to do this work, which is called ***ConstructsTraverser***. Its mechanism is also similar to ***GrammarASTTraverser***, and the difference is what is to be traversed, tree nodes in an AST or objects in the static ***ArrayList*** instance of each EBNF meta-concept class. Another difference is that ***GrammarASTTraverser*** creates objects and sets up associations according to the grammar of .g files, whereas ***ConstructsTraverser*** generates elements used in .uml files according to the objects and their associations, and also the grammar of .g files. For example, ***expr : proexpr (PLUS^ | MINUS^ proexpr)* SEMI!;*** this is a rule, *expr* is the rule reference ***NonTerminal*** object, which is associated with a *Composition* object standing for the right side with the multiplicity 1:1, as we have talked, what ***GrammarASTTraverser*** does is to create the objects and set up the association, whereas ***ConstructsTraverser*** is to generate `<class> expr <endclass>` and `<class> expr_main <endclass>`, and `+expr.<noName>[1] <-> +expr_main.<noName>[1];`. It seems not very complex, but I must explain some points.

First, if we let ***ConstructsTraverser*** simply generate the .uml contents as strings, which means the output is hardcoded, then the output content is mixed up with the output logic, so that the programming is error-prone, it has bad scalability and the code is not clear enough to read. Instead, I separated the traverser and the .uml contents to be output which were packed in some classes, look at the following picture:

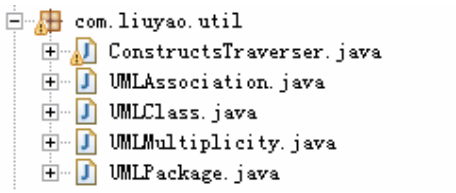

Fig 4.10 Some utility classes of Transformer

I defined classes for each of the main elements that are used in .uml files. All output content strings are packed in these classes. ***ConstructsTraverser***, at the right moment, only create objects for .uml elements with necessary information, such as class name to be inserted in between <*class*> and <*endclass*>, and multiplicities for associations. All these created objects standing for classes or associations are stored in two ***ArrayList*** instances in ***ConstructsTraverser***. When the final .uml file is to be written to the file system, we can simply call ***toString()*** method of each object.

Second, when ***ConstructsTraverser*** is traversing the EBNF meta-concepts objects, even each EBNF meta-concept class provides an ***ArrayList*** instance, it is still not convenient to use the variable ***allInstances*** of each class separately. Do we have to do it like that? Of course not. We need first explore the hierarchy of EBNF meta-concepts classes. Referring to Fig 2.1,

we should notice that class ***Expression*** is the superclass of all the others. This relationship should be reflected in the corresponding generated Java files, which were shown in Fig 3.13. The classes in package ***bnfmetamodel.internal.generated*** must implement the corresponding interfaces in package ***bnfmetamodel***; and the classes in package ***bnfmetamodel.internal*** must inherit the corresponding classes in package ***bnfmetamodel.internal.generated***. However, there is a special fact that class ***Expression*** is the superclass of the classes in the package ***bnfmetamodel.internal.generated*** only except class ***ExpressionGEN***, which means that other classes except class ***ExpressionGEN*** in package ***bnfmetamodel.internal.generated***, not only implement the interfaces in package ***bnfmetamodel***, but also inherit class ***Expression***. So we can draw the hierarchy of them in the following picture:

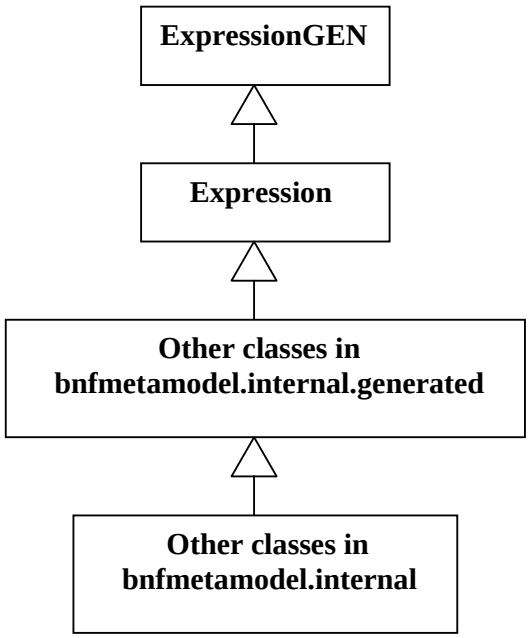

Fig 4.11 Hierarchy of EBNF meta-concepts classes

Besides this inheritance relationship, we would better also explore the constructors of these classes, and the following picture shows the constructors of four classes as an example:

```
public Alternative() {
    super();
}

public Expression() {
    super();
}
```

```
protected AlternativeGEN() {
    super();
    if ( usesAllInstances ) {
        allInstances.add(((Alternative)this));
    }
}
protected ExpressionGEN() {
    if ( usesAllInstances ) {
        allInstances.add(((Expression)this));
    }
}
```

Fig 4.12 Constructors of four EBNF meta-concepts classes

Therefore, when an ***Alternative*** object is created, the calling of constructors of superclasses is cascading. Finally, this object could be added to the static variable ***allInstances*** of class ***ExpressionGEN***. Thus we can simply use the static variable of class ***ExpressionGEN*** to get all EBNF meta-concepts objects, instead of using the static variable of each class in the same package as ***ExpressionGEN***.

Third, in traversing the EBNF meta-concepts objects, I involved a little optimization. In Fig 4.4, we can find some rules of .g files' grammar:

```
block   :   alternative ( alternative )* ;
ebnf    :    #( BLOCK block )
        |    #( OPTIONAL block )
        |    #( CLOSURE block )
        |    #( POSITIVE_CLOSURE block )
        ;
```

which means that *block*, *optional*, *closure* and *positive_closure* are all treated as block, and each block at least has an alternative. Thus, when meeting a tree node, which can be treated as a block, in an AST, an object of ***Composition***, ***Optional***, ***Arbitrary***, or ***AtLeastOne***, could be created for the tree node, and at least one object of ***Alternative*** is created to represent at least one alternative of the block. When generating the .uml file for the AST, the object of ***Alternative*** should be translated to a class in the .uml file, which inherits the class translated from that object which is standing for the block, Fig 4.13 gives an example, and *expr.uml* was transformed from *calc.g*, see Appendix IV.

```
<class> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0
            <specializes> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>

<class> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1
            <specializes> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>
```

Fig 4.13 A Fragment of *expr.uml*

However, if a tree node treated as a block only has one alternative, according to the .g files' grammar, ***GrammarASTTraverser*** would still create an ***Alternative*** object. Actually, it is meaningless and redundant to translate this ***Alternative*** object to a class which is specializing the block, because the only one alternative shouldn't be considered as an effective alternative. In this case, the optimization is necessary and ***ConstructsTraverser*** should be responsible for it. The procedure of it is not complex: every time before an block object of the four classes, ***Composition***, ***Optional***, ***Arbitrary***, or ***AtLeastOne***, is processed, I defined a method, which is named ***processAlternatives(Expression expr, List exprlist)***, to process its alternatives. This method is first to check how many alternatives this block object has as its subexpressions, if it has only one, all the subexpressions of this alternative are transferred into the block object as its subexpressions, and then this alternative is deleted from the subexpressions of the block object. Finally, appropriate associations between the class representing the block object and the classes representing the new subexpressions of the block object are properly set up. The code of class ***ConstructsTraverser*** is attached as Appendix II.

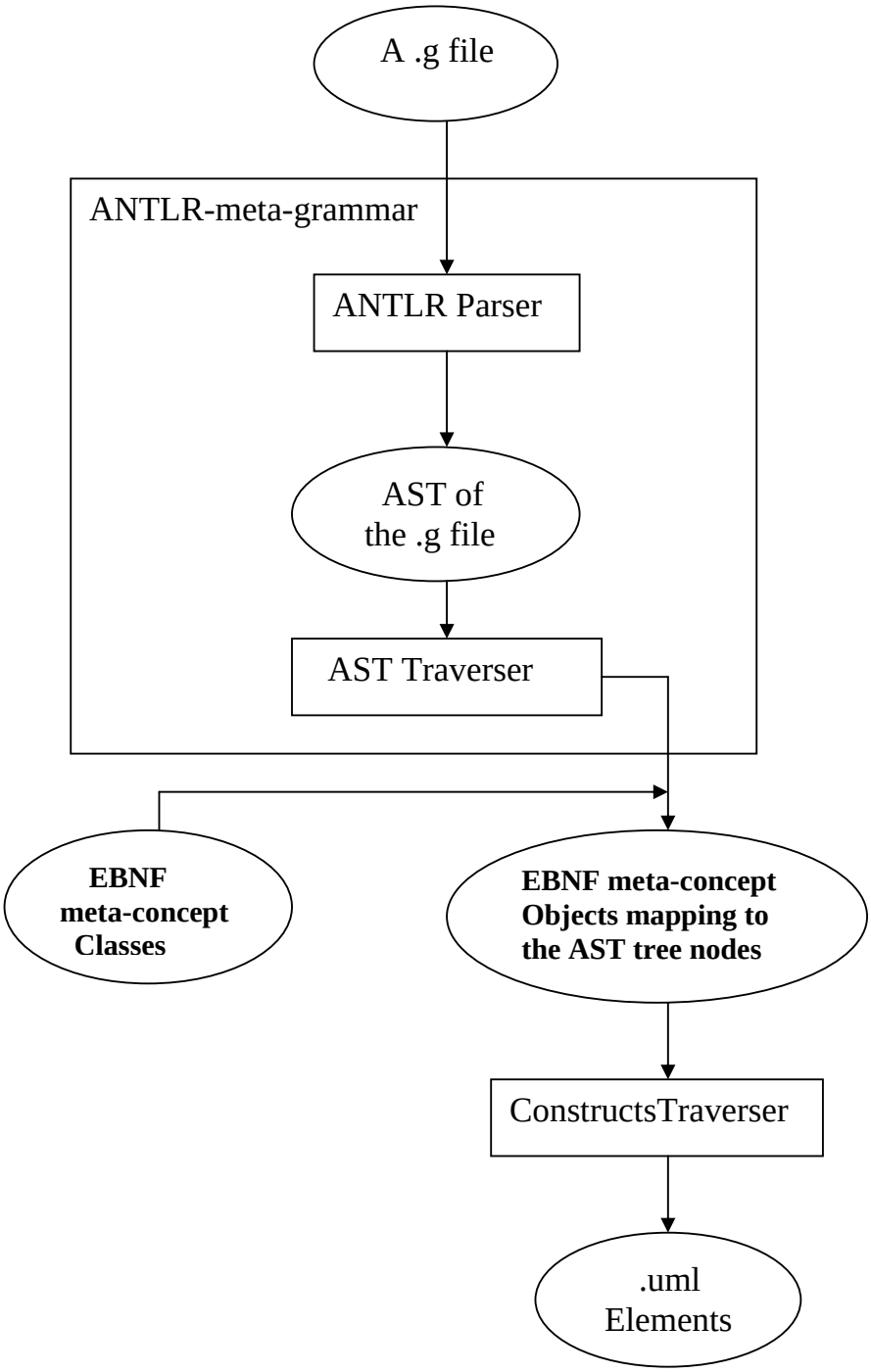

Fig 4.14 Workflow of Transformer

Fig 4.14 shows the entire workflow of our transformer, with the final result, .uml elements, the .uml file is easily generated. This transformer is language or grammar independent, so that it could be used to process almost any realistic .g files to .uml files, in Appendix V and Appendix VI are a realistic example ---- The grammar of micro-SQL in sql.g and its sql.uml generated by the transformer.

4.4 Implementing calc AST Traverser

Referring to Fig 4.1, we still have calc AST Traverser to implement. From the picture, we are clearly aware of its function: when a simple arithmetic calculation expression as input is parsed into an AST, calc AST Traverser traverses this AST and creates, for tree nodes in the AST, objects of the classes described in *expr.uml*. And these classes were generated by Octopus from *expr.uml*.

I defined a class ***CalcASTTraverser***. In fact, it is similar to ***GrammarASTTraverser***. So it is not necessary to explain every detail but some important points in the algorithm. And the code is available in Appendix III.

First of all, because this ***CalcASTTraverser*** is to walk in a tree, we would better get familiar with the structure features of calc AST trees. Here is a typical sample arithmetic expression: ***3+2*4-6^7+5/9;*** . Its AST is as the following picture:

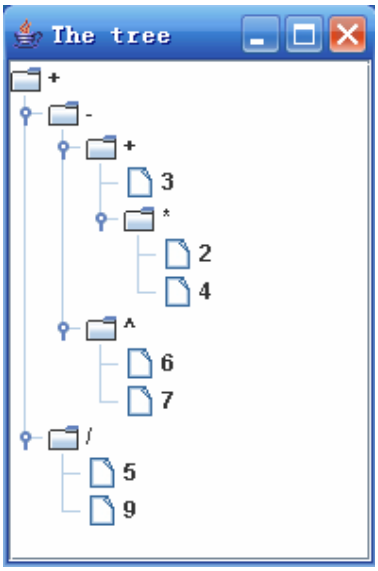

Fig 4.15 AST of ***3+2*4-6^7+5/9;***

In this picture, we can discover an interesting fact that, if a sum expression has several plus or minus operators, the subtrees with these sum operators as roots are always acting as the first child tree of their parent trees in the AST. For example, subtree (3+2*4) is the first child tree of its parent tree (3+2*4-6^7). And this also applies to multiply and divide operators. Moreover, the closer the numbers and operators appear to the beginning of an expression, the more inside they appear in its AST. Subexpression (3+2*4) is the first sum expression and appearing at the beginning of this expression, but in the AST, it is located at the core. In a sum subexpression, the closer the numbers and the multiply and divide operators appear to the beginning of it, the more inside they appear in the AST of that sum subexpression. After exploring many examples, the facts could be summarized to be true.

These features bring us much convenience when design the algorithm: plus and minus operators always appear in the first subtrees, and in a sum subexpression, multiply and divide

operators always appear in the first subtrees, too, therefore the first subtrees in calc ASTs are playing a different role from the second subtrees and mostly more important to the structure of the ASTs; and the operators in *calc.g* are all dual operators, so ***CalcASTTraverser*** can separately process two subtrees of each AST or sub AST. The methods for the first subtrees mainly focus on structural and procedural process, such as calling other methods, delegating processing tasks, and returning results, etc., whereas the methods for the second subtrees mainly focus on creating objects for tree nodes and adding identifying information.

Furthermore, the EBNF grammar in *calc.g* implies the priorities between the operators. Based on the priorities, a method chain was set up, and each method in it is responsible for a task. An AST tree node is passed from the top level to lower levels. And delegating tasks of processing subtrees is also handled in relative higher levels to lower levels.

Another point to mention is the output information stored in the objects of classes which were generated from *calc.uml*. I overwrote the method ***toString()*** in each of those classes. As among those classes are the associations described in *calc.uml*, it is easy to use in ***toString()*** the information from the objects of related classes. Fig 4.16 gives an example.

```
public class expr_main {
    private expr f_expr = null;
    private Set  f_expr_main_ALT_0_Arbitrary_0 = new HashSet();
    private proexpr f_proexpr = null;
    private SEMI f_sEMI = null;

    public String toString(){
        String result;
        result = "(" + f_proexpr.toString();
        Iterator iterator = f_expr_main_ALT_0_Arbitrary_0.iterator();
        while(iterator.hasNext()){
            result += ((expr_main_ALT_0_Arbitrary_0)iterator.next()).toString();
        }
        result += ")";
        return result;
    }
```

Fig 4.16 A Fragment of class ***expr_main***

4.5 Packing in a Plug-in of Eclipse

All logics in Fig 4.1 have been implemented, so it's time to pack them in a plug-in of Eclipse. In fact, I built this prototype unit by unit. When implementing each functional unit, I created a client for it. Now, we only need to build a simple plug-in to call those clients. It is much straightforward.

But I have to say this was not as easy as I thought, mostly because of the inherence of ANTLR-meta-grammar. You might have not understood why ANTLR-meta-grammar files are in the default package of Eclipse, without a named package declared, referring to Fig 4.2. Actually, I have tried to declare a package for it, but when it was executed, there was always

ClassCastException thrown out. It said that casting an ***AST*** object to ***GrammarAST*** caused this exception, but we have known that ***GrammarAST*** is a subclass of ***BaseAST*** that is just a subclass of ***AST***. I also checked the code that threw the exception, and I did not find that casting was not suitable. Furthermore, I tried in other ways, too, but finally, I had to leave it without a named package.

However, having not declared a package incurred a problem: Java classes can not be imported to other classes which has a package declared, simply by ***import*** statement. Even though I created a client for ANTLR-meta-grammar also in the default package, this client had still to be called in the plug-in when transforming a .g file to a .uml file. Therefore, this problem caused by no package declaration had to be solved ultimately.

Besides, I declared a package ***calc*** for the classes of calc Parser, referring to Fig 3.5. And we have built a client for calc Parser in Section 3.1.3, and its code is shown in Fig 3.6. This client should be called when parsing an input simple arithmetic expression. Unfortunately, it seems that Eclipse plug-in platform is strict to outside classes and methods which are to be called in a plug-in, so that a confusing event loop exception was always thrown out. It was even difficult to find out where this exception was thrown.

In a word, some generic drawbacks of ANTLR-mete-grammar did not allow for putting its classes under any packages, otherwise some error occurred; therefore, being in the default packages, its classes could not be imported into any other classes that were in declared packages; moreover, some incompatibilities between Eclipse plug-in platform and our own developed classes also caused errors.

Fortunately, I found out a way to solve this annoying problem by chance after trying with many different means. I found that Java reflection could bypass some incompatibilities and ***Class.forName(String classname)*** method could at runtime obtain the class you did not import, so that the classes in the default package could be used in any other classes in named packages and the incompatibilities did not come out any more.

```
try{
    Class c = Class.forName("MainTran");
    Method[] ms = c.getMethods();
    System.out.println(ms[0].getName());
    Object[] args = new Object[1];
    args[0] = path;
    for(int i = 0; i < ms.length; i++){
        if(ms[i].getName().equals("transform"))
            ms[i].invoke(null, args);
    }
}catch(Exception e){
    System.err.println(e.getMessage());
    e.printStackTrace();
}
```

Fig 4.17 Code Fragment for using Java Reflection to Call from outside a Plug-in

Fig 4.17 gives a code fragment of the plug-in for transformation. *MainTran* is the name of a class that is the client I built for ANTLR-meta-grammar. *transform* is the name of a method in that client. *path* is a string as an argument to be passed to *transform*.

After solving this problem, two functionalities should be merged into one single plug-in. This plug-in consists of a popup menu selection that only appears when right clicking a .g file and a view like a console where simple arithmetic expressions can be typed in, referring Fig 4.18 and Fig 4.19.

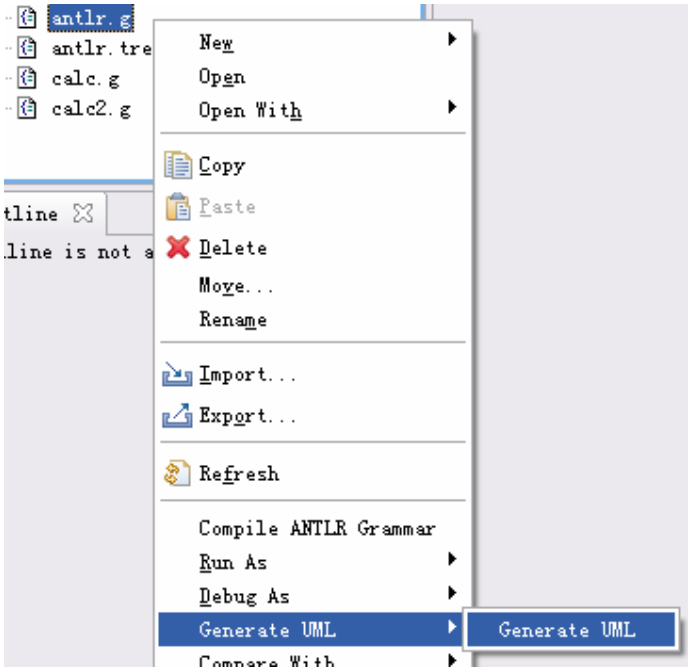

Fig 4.18 Popup Menu Selection for Transforming .g files

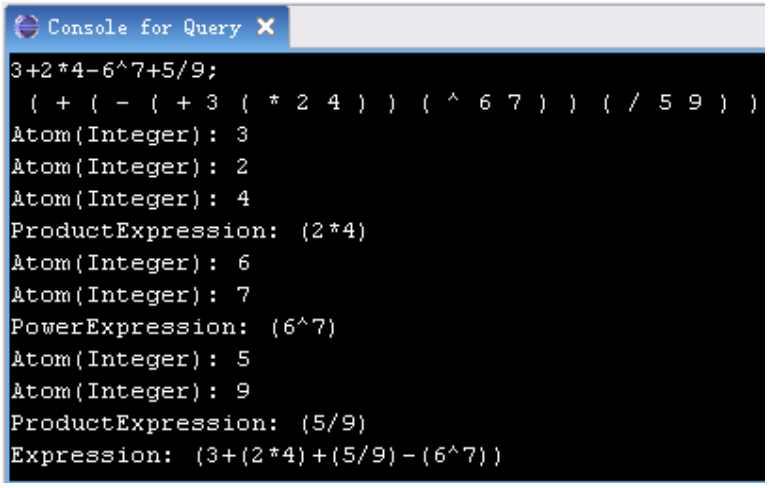

Fig 4.19 Console View for Parsing Simple Arithmetic Expressions
(This example was given in Section 4.4)

5 Conclusions

After exploring metamodeling approach for processing EBNF grammars, investigating ANTLR and Octopus with .g and .uml files' formats and grammars, and finally building the prototype, this project is finished.

As mentioned in Chapter 1, this project would process nRQL queries and serve JRacer. Now, a prototype is built and the entire workflow is successfully set up, which will definitely make the future work for nRQL easier. As is known to us, prototyping and spiraling are widely accepted and used in software development; therefore some further work should be delayed to the next cycle. The next step to enrich this project is to replace all stuff about simple arithmetic calculation expressions with nRQL, which means all *calc* in Fig 4.1 should be replaced by nRQL.

Because metamodeling approach was used, simple but inconvenient syntax-oriented EBNF grammars and the output simply generated by parsers can be changed to complex but convenient, flexible, and scalable object-oriented metamodels. Object-oriented metamodels are much more compatible with existing object-oriented frameworks, such as JRacer, etc. Moreover, it is convenient and not difficult to extend this prototype with other functionalities on handling nRQL syntax.

In the development, the idea of Model Driven Architecture was adopted. The fundamentals for transformation from one model to another are clearly described. The metamodelling approach and its transformation rules made it possible to use MDA as a reference. And it is this idea that promotes the automatic transformation from .g files to .uml files.

ANTLR plays an important role in this project. It is a powerful recognizer generator, and it can automatically generate a quasi-compiler, which can parse the original language and generate intermediate result but not the target code, for any languages with the grammar descriptions in .g files given. Besides, the developer also provides ANTLR-meta-grammar to parse .g files, which was helpful to this project.

The transformer is independent to languages which are to be transformed to object-oriented metamodels. This is the most important part in this prototype, and it also could be used elsewhere as an independent tool. In the future improvement of the prototype, it needs some optimizations, mainly on removing some unnecessary EBNF meta-concepts objects, which is similar to the optimization I have mentioned in Section 4.3.3, when creating .uml elements objects.

However, ***CalcASTTraverser*** is language independent. Actually, it was designed based on the EBNF grammar for simple arithmetic calculation expressions and their ASTs, thus such a traverser is designed particularly for a language and only working for it. This is, to some extent, a disadvantage. In order to process nRQL queries, a traverser only for nRQL is required to be newly designed and developed.

In conclusion, this prototype is completed and working well. And it provides a considerable start point and base for future work on nRQL.

References

[1] Information about Racer System *By Racer System GmbH*
http://www.racer-systems.com/products/racerpro/features.phtml
http://www.racer-systems.com/products/porter.phtml

[2] BNF and EBNF: What are they and how do they work? *By Lars Marius Garshol*
http://www.garshol.priv.no/download/text/bnf.html

[3] Metamodel *From Wikipeida, the free encyclopedia*
http://en.wikipedia.org/wiki/Metamodel

[4] A Metamodel for SDL-2000 in the Context of Metamodelling ULF
By Joachim Fischer, Michael Piefel and Markus Scheidgen
Institut fuer Informatik, Humboldt Universitaet zu Berlin

[5] An Introduction to ANTLR *By Terence Parr*
http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html

[6] ANTLR *By Ashley J.S Mills The University of Birmingham*
http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/antlr/antlr.html

[7] Information about Octopus *By Klasse Objecten, Soest, the Netherlands*
http://www.klasse.nl/english/research/octopus-intro.html

[8] Translators Should Use Tree Grammars
By Terence Parr University of San Francisco Nov 15, 2004

Appendix I

GrammarASTTraverser.java

```
import bnfmetamodel.internal.Alternative;
import bnfmetamodel.internal.Arbitrary;
import bnfmetamodel.internal.AtLeastOne;
import bnfmetamodel.internal.Composition;
import bnfmetamodel.internal.Expression;
import bnfmetamodel.internal.NonTerminal;
import bnfmetamodel.internal.Optional;
import bnfmetamodel.internal.Terminal;
import java.util.*;

public class GrammarASTTraverser implements ANTLRTokenTypes{
    private Stack blockStack;
    private int blockCounter = 0;
    private int altnCounter = 0;

    public GrammarASTTraverser(){
        this.blockStack = new Stack();
    }

    public GrammarAST traverseGrammarAST(GrammarAST ast) throws Exception{
        GrammarAST tempAst = ast;
        if(tempAst.getType() == PARSER){
            tempAst = (GrammarAST)tempAst.getFirstChild();
            if(tempAst == null) throw new Exception("No construct");
            else {
                while(tempAst != null && tempAst.getType() == RULE){
                    tempAst = traverseGrammarAST(tempAst);
                }
            }
        }
        else if(tempAst.getType() == RULE){
            tempAst = (GrammarAST)tempAst.getFirstChild();
           if(tempAst == null) throw new Exception("No construct");
           else{
               if(tempAst.getType() == RULE_REF){
                   NonTerminal ntrml = null;
                   List ntrmlList = NonTerminal.allInstances();
                   for(int i=0; i<ntrmlList.size(); i++){
                       String name = ((NonTerminal)ntrmlList.get(i)).getName();
                       if(name.equals(tempAst.getText())) {
                           ntrml = (NonTerminal)ntrmlList.get(i);
                           break;
                       }
                   }
                   if(ntrml == null){
                       ntrml = NonTerminal.newNonTerminal(tempAst.getText());
                   }
                   tempAst = (GrammarAST)tempAst.getNextSibling();
                   if(tempAst == null) throw new Exception("No construct");
```

```
                else{
                    if(tempAst.getType() == BLOCK){
                        Composition mainBlock =
                            Composition.newComposition(ntrml.getName()+"_main");
                        mainBlock.setNonTerminal(ntrml);
                        blockStack.push(mainBlock);
                        tempAst = traverseGrammarAST(tempAst);
                        blockStack.pop();
                    }
                    else throw new Exception("Unexpected construct");
                    if(tempAst.getType() != EOR)
                        throw new Exception("Unexpected construct");
                }
            }
            else throw new Exception("Unexpected construct");
        }
    }
    else if(tempAst.getType() == BLOCK || tempAst.getType() == OPTIONAL ||
            tempAst.getType() == CLOSURE || tempAst.getType() == POSITIVE_CLOSURE){
        Expression currentBlock = (Expression)blockStack.peek();
        int altcounter = 0;
        if(tempAst.getType() == OPTIONAL){
            String name = currentBlock.getName()+"_Optional_"+altnCounter;
            Optional optnl = Optional.newOptional(name);
            currentBlock.addToSubexpression(optnl);
            currentBlock = optnl;
        }else if(tempAst.getType() == CLOSURE){
            String name = currentBlock.getName()+"_Arbitrary_"+altnCounter;
            Arbitrary abtr = Arbitrary.newArbitrary(name);
            currentBlock.addToSubexpression(abtr);
            currentBlock = abtr;
        }else if(tempAst.getType() == POSITIVE_CLOSURE){
            String name = currentBlock.getName()+"_AtLeastOne_"+altnCounter;
            AtLeastOne alo = AtLeastOne.newAtLeastOne(name);
            currentBlock.addToSubexpression(alo);
            currentBlock = alo;
        }else if(tempAst.getType() == BLOCK && currentBlock instanceof Alternative){
            String name = currentBlock.getName()+"_Block_"+altnCounter;
            Composition cmpt = Composition.newComposition(name);
            currentBlock.addToSubexpression(cmpt);
            currentBlock = cmpt;
        }
        tempAst = (GrammarAST)tempAst.getFirstChild();
        if(tempAst == null) throw new Exception("No construct");
        else{
            while(tempAst != null && tempAst.getType() == ALT){
                Alternative alt = Alternative.newAlternative(currentBlock.getName() +
                                  "_ALT_" + altcounter++);
                currentBlock.addToSubexpression(alt);
                blockStack.push(alt);
                tempAst = traverseGrammarAST(tempAst);
                blockStack.pop();
            }
        }
    }
    else if(tempAst.getType() == ALT){
        int subAltCounter = 0;
        tempAst = (GrammarAST)tempAst.getFirstChild();
        if(tempAst == null) throw new Exception("No construct");
        else{
            int type = tempAst.getType();
            while(type == BLOCK || type == OPTIONAL || type == CLOSURE ||
```

```
                type == POSITIVE_CLOSURE || type == RULE_REF || type == TOKEN_REF ||
                type == CHAR_LITERAL || type == STRING_LITERAL || type == WILDCARD){
            tempAst = traverseGrammarAST(tempAst);
            if(tempAst == null) break;
            type = tempAst.getType();
            altnCounter = subAltCounter++;
        }
    }
}
else if(tempAst.getType() == RULE_REF){
    NonTerminal ntrml = null;
    List ntrmlList = NonTerminal.allInstances();
    for(int i=0; i<ntrmlList.size(); i++){
        String name = ((NonTerminal)ntrmlList.get(i)).getName();
        if(name.equals(tempAst.getText())) {
            ntrml = (NonTerminal)ntrmlList.get(i);
            break;
        }
    }
    if(ntrml == null){
        ntrml = NonTerminal.newNonTerminal(tempAst.getText());
    }
    Alternative aln = (Alternative)blockStack.peek();
    aln.addToSubexpression(ntrml);
}
else if(tempAst.getType() == TOKEN_REF){
    Terminal trml = null;
    List trmlList = Terminal.allInstances();
    for(int i=0; i<trmlList.size(); i++){
        String name = ((Terminal)trmlList.get(i)).getName();
        if(name.equals(tempAst.getText())) {
                trml = (Terminal)trmlList.get(i);
                break;
            }
        }
        if(trml == null){
            trml = Terminal.newTerminal(tempAst.getText());
        }
        Alternative aln = (Alternative)blockStack.peek();
        aln.addToSubexpression(trml);
  }
  else if(tempAst.getType() == CHAR_LITERAL){
        Terminal trml = null;
        List trmlList = Terminal.allInstances();
        for(int i=0; i<trmlList.size(); i++){
            String name = ((Terminal)trmlList.get(i)).getName();
            if(name.equals(tempAst.getText())) {
                trml = (Terminal)trmlList.get(i);
                break;
            }
        }
        if(trml == null){
            trml = Terminal.newTerminal(tempAst.getText());
        }
        Alternative aln = (Alternative)blockStack.peek();
        aln.addToSubexpression(trml);
  }
  else if(tempAst.getType() == STRING_LITERAL){
        Terminal trml = null;
        List trmlList = Terminal.allInstances();
```

```
            for(int i=0; i<trmlList.size(); i++){
                String name = ((Terminal)trmlList.get(i)).getName();
                if(name.equals(tempAst.getText())) {
                    trml = (Terminal)trmlList.get(i);
                    break;
                }
            }
            if(trml == null){
                trml = Terminal.newTerminal(tempAst.getText());
            }
            Alternative aln = (Alternative)blockStack.peek();
            aln.addToSubexpression(trml);
        }
        else if(tempAst.getType() == WILDCARD){
            Terminal trml = null;
            List trmlList = Terminal.allInstances();
            for(int i=0; i<trmlList.size(); i++){
                String name = ((Terminal)trmlList.get(i)).getName();
                if(name.equals(tempAst.getText())) {
                    trml = (Terminal)trmlList.get(i);
                    break;
                }
            }
            if(trml == null){
                trml = Terminal.newTerminal(tempAst.getText());
            }
            Alternative aln = (Alternative)blockStack.peek();
            aln.addToSubexpression(trml);
        }
        return (GrammarAST)ast.getNextSibling();
    }
}
```

Appendix II

ConstructsTraverser.java

```
package com.liuyao.util;
import bnfmetamodel.internal.*;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Set;
import java.util.Iterator;

public class ConstructsTraverser {
    private ArrayList classList = null;
    private ArrayList assList = null;
    private String packageName = null;

    public ConstructsTraverser(){
        classList = new ArrayList();
        assList = new ArrayList();
    }

    public static void clearAllConstructs(){
        Alternative.allInstances().clear();
        Arbitrary.allInstances().clear();
        AtLeastOne.allInstances().clear();
        Composition.allInstances().clear();
        Expression.allInstances().clear();
        NonTerminal.allInstances().clear();
        Optional.allInstances().clear();
        Symbol.allInstances().clear();
        Terminal.allInstances().clear();
    }

    public ArrayList getClassList(){
        return classList;
    }

    public ArrayList getAssociationList(){
        return assList;
    }

    public String getPackageName(){
        return packageName;
    }

    private boolean processAlternatives(Expression expr, List exprlist){
        Collection altnCollection;
        if(expr instanceof Composition)
            altnCollection = ((Composition)expr).getSubexpression();
        else if(expr instanceof Optional)
            altnCollection = ((Optional)expr).getSubexpression();
        else if(expr instanceof Arbitrary)
            altnCollection = ((Arbitrary)expr).getSubexpression();
```

```
    else if(expr instanceof AtLeastOne)
        altnCollection = ((AtLeastOne)expr).getSubexpression();
    else return false;

        if(altnCollection.size() > 1){
            Iterator iter = altnCollection.iterator();
            while(iter.hasNext()){
                classList.add(new UMLClass(((Alternative)iter.next()).getName(),
                        expr.getName()));
            }
            return false;
        } else if(altnCollection.size() == 1){
            Iterator iter = altnCollection.iterator();
            Alternative toBeDeleted = (Alternative)iter.next();
            Set subexprSet = toBeDeleted.getSubexpression();
            expr.removeAllFromSubexpression();
            exprlist.remove(toBeDeleted);
            Iterator subIter = subexprSet.iterator();
            while(subIter.hasNext()){
                expr.addToSubexpression((Expression)subIter.next());
            }
            return true;
        }
        else return false;
}

private void furtherProcessForAlternatives(Expression expr){
    Collection subexprCollection;
    if(expr instanceof Composition)
        subexprCollection = ((Composition)expr).getSubexpression();
    else if(expr instanceof Optional)
         subexprCollection = ((Optional)expr).getSubexpression();
     else if(expr instanceof Arbitrary)
         subexprCollection = ((Arbitrary)expr).getSubexpression();
     else if(expr instanceof AtLeastOne)
         subexprCollection = ((AtLeastOne)expr).getSubexpression();
     else if(expr instanceof Alternative)
         subexprCollection = ((Alternative)expr).getSubexpression();
     else return;

     Iterator iter = subexprCollection.iterator();
     while(iter.hasNext()){
         Expression subexpr = (Expression)iter.next();
         UMLMultiplicity multi = UMLMultiplicity.getMulti(subexpr);
         assList.add(new UMLAssociation(expr.getName(), subexpr.getName(),
                 UMLMultiplicity.SINGLE, multi));
     }
}

public void traverseConstructs()throws Exception{
     List exprList = Expression.allInstances();
     if(exprList.isEmpty()) return;
     Symbol classExtendsParser = (Symbol)exprList.get(0);
     packageName = classExtendsParser.getName();
     for(int i = 0; i < exprList.size(); i++){
         Expression expr = (Expression)exprList.get(i);
         if(expr instanceof Alternative){
             furtherProcessForAlternatives(expr);
         }else if(expr instanceof NonTerminal){
```

```
                classList.add(new UMLClass(expr.getName()));
            }else if(expr instanceof Terminal){
                classList.add(new UMLClass(expr.getName()));
            }else if(expr instanceof Symbol){
                throw new Exception("Wrong construct--Symbol encountered.");
            }else {
                classList.add(new UMLClass(expr.getName()));
                if(expr instanceof Composition){
                    if(expr.getNonTerminal()!=null){
                        NonTerminal ntrml = (NonTerminal)expr.getNonTerminal();
                        assList.add(new UMLAssociation(ntrml.getName(), expr.getName(),
                                UMLMultiplicity.SINGLE, UMLMultiplicity.SINGLE));
                    }
                    if(processAlternatives(expr, exprList)){
                        furtherProcessForAlternatives(expr);
                    }
                }else if(expr instanceof Optional){
                    if(processAlternatives(expr, exprList)){
                        furtherProcessForAlternatives(expr);
                    }
                }else if(expr instanceof Arbitrary){
                    if(processAlternatives(expr, exprList)){
                        furtherProcessForAlternatives(expr);
                    }
                }else if(expr instanceof AtLeastOne){
                    if(processAlternatives(expr, exprList)){
                        furtherProcessForAlternatives(expr);
                    }
                }
            }
        }
    }
}
```

Appendix III

CalcATSTraverser.java

```
package calc;
import expr.*;
import antlr.collections.AST;

public class CalcASTTraverser implements CalcTreeWalkerTokenTypes{
    private boolean has_main_expr = false;
    private boolean has_main_proexpr = false;
    private expr main = null;
    private proexpr main_proexpr = null;
    private int exprCounter = 0;
    private int proexprCounter = 0;
    private String output;
    public CalcASTTraverser(){
        output = "";
    }

    public String traverseCalcAST(AST calcAST) throws Exception{
        AST tempAST = calcAST;
        if(tempAST.getType() == PLUS || tempAST.getType() == MINUS){
            traverseExpr(tempAST);
        }
        else if(tempAST.getType() == MUL || tempAST.getType() == DIV){
            traverseProexpr(tempAST);
        }
        else if(tempAST.getType() == POW){
            traversePowpr(tempAST);
        }
        else if(tempAST.getType() == INT){
            traverseAtom(tempAST);
        }
        else throw new Exception("Unexpected AST type");
       return output;
    }

    private expr traverseExpr(AST exprAST){
        expr temp = null;
        AST firstProexpr = exprAST.getFirstChild();
        AST secondProexpr = firstProexpr.getNextSibling();
        if(firstProexpr.getType() == PLUS || firstProexpr.getType() == MINUS){
            has_main_proexpr = false;
            exprCounter++;
            temp = traverseExpr(firstProexpr);
            exprCounter--;
            traverseSecondSubExpr(exprAST, secondProexpr, temp);
            if(exprCounter == 0)
                output += "Expression: "+temp.toString()+"\n";
            return temp;
        }
```

```
    if(!has_main_expr){
        main = new expr();
        expr_main exprMain = new expr_main();
        main.setExpr_main(exprMain);
        has_main_proexpr = false;
        exprMain.setProexpr(traverseProexpr(firstProexpr));
        temp = main;
        has_main_expr = true;
    }
    traverseSecondSubExpr(exprAST, secondProexpr, temp);
    if(exprCounter == 0)
        output += "Expression: "+temp.toString()+"\n";
    return temp;
}

private void traverseSecondSubExpr
        (AST exprAST, AST secondProexpr, expr parentOfSecondSubExpr){
    expr_main_ALT_0_Arbitrary_0 arbitrary_0 = new expr_main_ALT_0_Arbitrary_0();
    expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0 alt_block = null;
    if(exprAST.getType() == PLUS){
        alt_block = new expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0();
        ((expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0)alt_block).setPLUS(new PLUS());
    }
    else {
        alt_block = new expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1();
        ((expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1)alt_block).setMINUS(new MINUS());
    }
    arbitrary_0.setExpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0(alt_block);
    has_main_proexpr = false;
    arbitrary_0.setProexpr(traverseProexpr(secondProexpr));
    parentOfSecondSubExpr.getExpr_main().addToExpr_main_ALT_0_Arbitrary_0(arbitrary_0);
}

private proexpr traverseProexpr(AST proexprAST){
    proexpr temp = null;
    if(proexprAST.getType() == POW || proexprAST.getType() == INT){
        temp = new proexpr();
        proexpr_main proexprMain = new proexpr_main();
        temp.setProexpr_main(proexprMain);
        proexprMain.setPowpr(traversePowpr(proexprAST));
        return temp;
    }
    AST firstPowexpr = proexprAST.getFirstChild();
    AST secondPowexpr = firstPowexpr.getNextSibling();
    if(firstPowexpr.getType() == MUL || firstPowexpr.getType() == DIV){
        proexprCounter++;
        temp = traverseProexpr(firstPowexpr);
        proexprCounter--;
        traverseSecondSubProexpr(proexprAST, secondPowexpr, temp);
        if(proexprCounter == 0)
            output += "ProductExpression: "+temp.toString()+"\n";
        return temp;
    }
    if(!has_main_proexpr){
        main_proexpr = new proexpr();
        proexpr_main proexprMain = new proexpr_main();
        main_proexpr.setProexpr_main(proexprMain);
        proexprMain.setPowpr(traversePowpr(firstPowexpr));
        temp = main_proexpr;
        has_main_proexpr = true;
    }
    traverseSecondSubProexpr(proexprAST, secondPowexpr, temp);
    if(proexprCounter == 0)
        output += "ProductExpression: "+temp.toString()+"\n";
    return temp;
}

private void traverseSecondSubProexpr
(AST proexprAST, AST secondPowexpr, proexpr parentOfSecondSubProexpr){
    proexpr_main_ALT_0_Arbitrary_0 arbitrary_0 = new proexpr_main_ALT_0_Arbitrary_0();
```

```
        proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0 alt_block = null;
        if(proexprAST.getType() == MUL){
            alt_block = new proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0();
            ((proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0)alt_block).setMUL(new MUL());
        }
        else {
            alt_block = new proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1();
            ((proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1)alt_block).setDIV(new DIV());
        }
        arbitrary_0.setProexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0(alt_block);
        arbitrary_0.setPowpr(traversePowpr(secondPowexpr));
        parentOfSecondSubProexpr.getProexpr_main().addToProexpr_main_ALT_0_Arbitrary_0(
                arbitrary_0);
    }

    private powpr traversePowpr(AST powprAST){
        powpr temp = null;
        if(powprAST.getType() == INT){
            temp = new powpr();
            powpr_main powexprMain = new powpr_main();
            temp.setPowpr_main(powexprMain);
            powexprMain.setAtom(traverseAtom(powprAST));
            return temp;
        }
        AST firstAtom = powprAST.getFirstChild();
        AST secondAtom = firstAtom.getNextSibling();
        temp = new powpr();
        powpr_main powexprMain = new powpr_main();
        temp.setPowpr_main(powexprMain);
        powexprMain.setAtom(traverseAtom(firstAtom));
        powpr_main_ALT_0_Optional_0 optional_0 = new powpr_main_ALT_0_Optional_0();
        optional_0.setPOW(new POW());
        optional_0.setAtom(traverseAtom(secondAtom));
        powexprMain.setPowpr_main_ALT_0_Optional_0(optional_0);
        output += "PowerExpression: "+temp.toString()+"\n";
        return temp;
    }

    private atom traverseAtom(AST atomAST){
        atom temp = null;
        temp = new atom();
        atom_main atom_main = new atom_main();
        temp.setAtom_main(atom_main);
        INT num = new INT();
        num.setValue(Integer.parseInt(atomAST.getText()));
        atom_main.setINT(num);
        output += "Atom(Integer): "+temp.toString()+"\n";
        return temp;
    }
}
```

Appendix IV

expr.uml

```
<package>expr

<class> expr
<endclass>

<class> expr_main
<endclass>

<class> proexpr
<endclass>

<class> expr_main_ALT_0_Arbitrary_0
<endclass>

<class> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>

<class> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0
    <specializes> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>

<class> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1
    <specializes> expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>

<class> PLUS
<endclass>

<class> MINUS
<endclass>

<class> SEMI
<endclass>

<class> proexpr_main
<endclass>

<class> powpr
<endclass>

<class> proexpr_main_ALT_0_Arbitrary_0
<endclass>

<class> proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>

<class> proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0
    <specializes> proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>

<class> proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1
    <specializes> proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0
<endclass>
```

```
<class> MUL
<endclass>

<class> DIV
<endclass>

<class> powpr_main
<endclass>

<class> atom
<endclass>

<class> powpr_main_ALT_0_Optional_0
<endclass>

<class> POW
<endclass>

<class> atom_main
<endclass>

<class> INT
<endclass>

<associations>
+expr.<noName>[1] <-> +expr_main.<noName>[1];
+expr_main.<noName>[1] <-> +expr_main_ALT_0_Arbitrary_0.<noName>[0..*];
+expr_main.<noName>[1] <-> +proexpr.<noName>[1];
+expr main.<noName>[1] <-> +SEMI.<noName>[1];
+expr_main_ALT_0_Arbitrary_0.<noName>[1] <->
                    +expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0.<noName>[1];
+expr_main_ALT_0_Arbitrary_0.<noName>[1] <-> +proexpr.<noName>[1];
+expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0.<noName>[1] <-> +PLUS.<noName>[1];
+expr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1.<noName>[1] <-> +MINUS.<noName>[1];
+proexpr.<noName>[1] <-> +proexpr_main.<noName>[1];
+proexpr_main.<noName>[1] <-> +proexpr_main_ALT_0_Arbitrary_0.<noName>[0..*];
+proexpr_main.<noName>[1] <-> +powpr.<noName>[1];
+proexpr_main_ALT_0_Arbitrary_0.<noName>[1] <->
                    +proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0.<noName>[1];
+proexpr_main_ALT_0_Arbitrary_0.<noName>[1] <-> +powpr.<noName>[1];
+proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_0.<noName>[1] <-> +MUL.<noName>[1];
+proexpr_main_ALT_0_Arbitrary_0_ALT_0_Block_0_ALT_1.<noName>[1] <-> +DIV.<noName>[1];
+powpr.<noName>[1] <-> +powpr_main.<noName>[1];
+powpr_main.<noName>[1] <-> +powpr_main_ALT_0_Optional_0.<noName>[0..1];
+powpr_main.<noName>[1] <-> +atom.<noName>[1];
+powpr_main_ALT_0_Optional_0.<noName>[1] <-> +POW.<noName>[1];
+powpr_main_ALT_0_Optional_0.<noName>[1] <-> +atom.<noName>[1];
+atom.<noName>[1] <-> +atom_main.<noName>[1];
+atom_main.<noName>[1] <-> +INT.<noName>[1];

<endpackage>
```

Appendix V

sql.g

```
class SQLParser extends Parser;

sql : (command SEMI!)+ ;

command : create_table
        | drop_table
        | insert_row
        | delete_rows
        | query
        | bye
        ;

create_table : "CREATE" "TABLE" table_name "("
                                    def_column_list
                                    "PRIMARY" "KEY" "(" key_list ")"
                                ")" ;

def_column_list : ( column_name column_type "," )+ ;

key_list : column_name ( "," column_name )* ;

drop_table : "DROP" "TABLE" table_name ;

table_name : IDENTIFIER ;

column_name : IDENTIFIER ;

column type : ( "INTEGER" | "VARCHAR" ) "(" NUMBER ")" ;

insert_row : ("INSERT" "INTO") table_name "VALUES" "(" constant ( "," constant )* ")"
           ;

delete_rows : ("DELETE" "FROM") table_name ( alias )? ( "WHERE" comparison )?
            ;

alias : IDENTIFIER ;

comparison : value COMP_OP^ value ;

value : ( alias "." )? column_name | constant ;

constant : NUMBER | STRING ;

query : "SELECT" ( "DISTINCT" )? query_column_list
                 "FROM" table_list
                 ( "WHERE" condition )
      ;

query_column_list : ( alias "." )? column_name ( "," ( alias "." )? column_name )* | "*"
                  ;

table_list : table_name ( alias )? ( "," table_name ( alias )? )* ;

condition : and_term ( "OR" condition )? ;

and_term : comparison ( "AND" and_term )? ;

bye : "QUIT" ;
```

```
class SQLLexer extends Lexer;

SEMI : ";" ;

COMP_OP : "=" | "<" | ">" | "<=" | ">=" | "<>" ;

DIGIT : "0".."9" ;

LETTER : "A".."Z" | "a".."z" ;

STRING : "'" ( " ".."&" | "(".."~" )* "'" ;

NUMBER : ( DIGIT )+ ;

IDENTIFIER : LETTER ( LETTER | DIGIT | "_" )* ;
```

Appendix VI

sql.uml

```
<package>sql

<class> sql
<endclass>

<class> sql_main
<endclass>

<class> sql_main_ALT_0_AtLeastOne_0
<endclass>

<class> command
<endclass>

<class> SEMI
<endclass>

<class> command_main
<endclass>

<class> command_main_ALT_0 <specializes> command_main
<endclass>

<class> command_main_ALT_1 <specializes> command_main
<endclass>

<class> command_main_ALT_2 <specializes> command_main
<endclass>

<class> command_main_ALT_3 <specializes> command_main
<endclass>

<class> command_main_ALT_4 <specializes> command_main
<endclass>

<class> command_main_ALT_5 <specializes> command_main
<endclass>

<class> create_table
<endclass>

<class> drop_table
<endclass>

<class> insert_row
<endclass>

<class> delete_rows
<endclass>

<class> query
<endclass>

<class> bye
<endclass>

<class> create_table_main
<endclass>

<class> "CREATE"
<endclass>

<class> "TABLE"
<endclass>
```

```
<class> table_name
<endclass>

<class> "("
<endclass>

<class> def_column_list
<endclass>

<class> "PRIMARY"
<endclass>

<class> "KEY"
<endclass>

<class> key_list
<endclass>

<class> ")"
<endclass>

<class> def_column_list_main
<endclass>

<class> def_column_list_main_ALT_0_AtLeastOne_9
<endclass>

<class> column_name
<endclass>

<class> column_type
<endclass>

<class> ","
<endclass>

<class> key_list_main
<endclass>

<class> key_list_main_ALT_0_Arbitrary_0
<endclass>

<class> drop_table_main
<endclass>

<class> "DROP"
<endclass>

<class> table_name_main
<endclass>

<class> IDENTIFIER
<endclass>

<class> column_name_main
<endclass>

<class> column_type_main
<endclass>

<class> column_type_main_ALT_0_Block_1
<endclass>

<class> column_type_main_ALT_0_Block_1_ALT_0 <specializes> column_type_main_ALT_0_Block_1
<endclass>
```

```
<class> column_type_main_ALT_0_Block_1_ALT_1 <specializes> column_type_main_ALT_0_Block_1
<endclass>

<class> "INTEGER"
<endclass>

<class> "VARCHAR"
<endclass>

<class> NUMBER
<endclass>

<class> insert_row_main
<endclass>

<class> insert_row_main_ALT_0_Block_2
<endclass>

<class> "INSERT"
<endclass>

<class> "INTO"
<endclass>

<class> "VALUES"
<endclass>

<class> constant
<endclass>

<class> insert_row_main_ALT_0_Arbitrary_4
<endclass>

<class> delete_rows_main
<endclass>

<class> delete_rows_main_ALT_0_Block_5
<endclass>

<class> "DELETE"
<endclass>

<class> "FROM"
<endclass>

<class> delete_rows_main_ALT_0_Optional_1
<endclass>

<class> alias
<endclass>

<class> delete_rows_main_ALT_0_Optional_2
<endclass>

<class> "WHERE"
<endclass>

<class> comparison
<endclass>

<class> alias_main
<endclass>

<class> comparison_main
<endclass>
```

```
<class> value
<endclass>

<class> COMP_OP
<endclass>

<class> value_main
<endclass>

<class> value_main_ALT_0 <specializes> value_main
<endclass>

<class> value_main_ALT_1 <specializes> value_main
<endclass>

<class> value_main_ALT_0_Optional_1
<endclass>

<class> "."
<endclass>

<class> constant_main
<endclass>

<class> constant_main_ALT_0 <specializes> constant_main
<endclass>

<class> constant_main_ALT_1 <specializes> constant_main
<endclass>

<class> STRING
<endclass>

<class> query_main
<endclass>

<class> "SELECT"
<endclass>

<class> query_main_ALT_0_Optional_0
<endclass>

<class> "DISTINCT"
<endclass>

<class> query_column_list
<endclass>

<class> table_list
<endclass>

<class> query_main_ALT_0_Block_4
<endclass>

<class> condition
<endclass>

<class> query_column_list_main
<endclass>

<class> query_column_list_main_ALT_0 <specializes> query_column_list_main
<endclass>

<class> query_column_list_main_ALT_1 <specializes> query_column_list_main
<endclass>
```

```
<class> query_column_list_main_ALT_0_Optional_0
<endclass>

<class> query_column_list_main_ALT_0_Arbitrary_1
<endclass>

<class> query_column_list_main_ALT_0_Arbitrary_1_ALT_0_Optional_0
<endclass>

<class> "*"
<endclass>

<class> table_list_main
<endclass>

<class> table_list_main_ALT_0_Optional_0
<endclass>

<class> table_list_main_ALT_0_Arbitrary_1
<endclass>

<class> table_list_main_ALT_0_Arbitrary_1_ALT_0_Optional_1
<endclass>

<class> condition_main
<endclass>

<class> and_term
<endclass>

<class> condition_main_ALT_0_Optional_0
<endclass>

<class> "OR"
<endclass>

<class> and_term_main
<endclass>

<class> and_term_main_ALT_0_Optional_0
<endclass>

<class> "AND"
<endclass>

<class> bye_main
<endclass>

<class> "QUIT"
<endclass>

<associations>
+sql.<noName>[1] <-> +sql_main.<noName>[1];
+sql_main.<noName>[1] <-> +sql_main_ALT_0_AtLeastOne_0.<noName>[1..*];
+sql_main_ALT_0_AtLeastOne_0.<noName>[1] <-> +command.<noName>[1];
+sql_main_ALT_0_AtLeastOne_0.<noName>[1] <-> +SEMI.<noName>[1];
+command.<noName>[1] <-> +command_main.<noName>[1];
+command_main_ALT_0.<noName>[1] <-> +create_table.<noName>[1];
+command_main_ALT_1.<noName>[1] <-> +drop_table.<noName>[1];
+command_main_ALT_2.<noName>[1] <-> +insert_row.<noName>[1];
+command_main_ALT_3.<noName>[1] <-> +delete_rows.<noName>[1];
+command_main_ALT_4.<noName>[1] <-> +query.<noName>[1];
+command_main_ALT_5.<noName>[1] <-> +bye.<noName>[1];
+create_table.<noName>[1] <-> +create_table_main.<noName>[1];
+create_table_main.<noName>[1] <-> +table_name.<noName>[1];
```

```
+create_table_main.<noName>[1] <-> +def_column_list.<noName>[1];
+create_table_main.<noName>[1] <-> +")".<noName>[1];
+create_table_main.<noName>[1] <-> +"CREATE".<noName>[1];
+create_table_main.<noName>[1] <-> +"PRIMARY".<noName>[1];
+create_table_main.<noName>[1] <-> +"(".<noName>[1];
+create_table_main.<noName>[1] <-> +key_list.<noName>[1];
+create_table_main.<noName>[1] <-> +"KEY".<noName>[1];
+create_table_main.<noName>[1] <-> +"TABLE".<noName>[1];
+def_column_list.<noName>[1] <-> +def_column_list_main.<noName>[1];
+def_column_list_main.<noName>[1] <-> +def_column_list_main_ALT_0_AtLeastOne_9.<noName>[1..*];
+def_column_list_main_ALT_0_AtLeastOne_9.<noName>[1] <-> +column_type.<noName>[1];
+def_column_list_main_ALT_0_AtLeastOne_9.<noName>[1] <-> +column_name.<noName>[1];
+def_column_list_main_ALT_0_AtLeastOne_9.<noName>[1] <-> +",".<noName>[1];
+key_list.<noName>[1] <-> +key_list_main.<noName>[1];
+key_list_main.<noName>[1] <-> +key_list_main_ALT_0_Arbitrary_0.<noName>[0..*];
+key_list_main.<noName>[1] <-> +column_name.<noName>[1];
+key_list_main_ALT_0_Arbitrary_0.<noName>[1] <-> +column_name.<noName>[1];
+key_list_main_ALT_0_Arbitrary_0.<noName>[1] <-> +",".<noName>[1];
+drop_table.<noName>[1] <-> +drop_table_main.<noName>[1];
+drop_table_main.<noName>[1] <-> +table_name.<noName>[1];
+drop_table_main.<noName>[1] <-> +"TABLE".<noName>[1];
+drop_table_main.<noName>[1] <-> +"DROP".<noName>[1];
+table_name.<noName>[1] <-> +table_name_main.<noName>[1];
+table_name_main.<noName>[1] <-> +IDENTIFIER.<noName>[1];
+column_name.<noName>[1] <-> +column_name_main.<noName>[1];
+column_name_main.<noName>[1] <-> +IDENTIFIER.<noName>[1];
+column_type.<noName>[1] <-> +column_type_main.<noName>[1];
+column_type_main.<noName>[1] <-> +NUMBER.<noName>[1];
+column_type_main.<noName>[1] <-> +")".<noName>[1];
+column_type_main.<noName>[1] <-> +"(".<noName>[1];
+column_type_main.<noName>[1] <-> +column_type_main_ALT_0_Block_1.<noName>[1];
+column_type_main_ALT_0_Block_1_ALT_0.<noName>[1] <-> +"INTEGER".<noName>[1];
+column_type_main_ALT_0_Block_1_ALT_1.<noName>[1] <-> +"VARCHAR".<noName>[1];
+insert_row.<noName>[1] <-> +insert_row_main.<noName>[1];
+insert_row_main.<noName>[1] <-> +"VALUES".<noName>[1];
+insert_row_main.<noName>[1] <-> +table_name.<noName>[1];
+insert_row_main.<noName>[1] <-> +")".<noName>[1];
+insert_row_main.<noName>[1] <-> +constant.<noName>[1];
+insert_row_main.<noName>[1] <-> +"(".<noName>[1];
+insert_row_main.<noName>[1] <-> +insert_row_main_ALT_0_Arbitrary_4.<noName>[0..*];
+insert_row_main.<noName>[1] <-> +insert_row_main_ALT_0_Block_2.<noName>[1];
+insert_row_main_ALT_0_Block_2.<noName>[1] <-> +"INTO".<noName>[1];
+insert_row_main_ALT_0_Block_2.<noName>[1] <-> +"INSERT".<noName>[1];
+insert_row_main_ALT_0_Arbitrary_4.<noName>[1] <-> +constant.<noName>[1];
+insert_row_main_ALT_0_Arbitrary_4.<noName>[1] <-> +",".<noName>[1];
+delete_rows.<noName>[1] <-> +delete_rows_main.<noName>[1];
+delete_rows_main.<noName>[1] <-> +delete_rows_main_ALT_0_Optional_1.<noName>[0..1];
+delete_rows_main.<noName>[1] <-> +table_name.<noName>[1];
+delete_rows_main.<noName>[1] <-> +delete_rows_main_ALT_0_Block_5.<noName>[1];
+delete_rows_main.<noName>[1] <-> +delete_rows_main_ALT_0_Optional_2.<noName>[0..1];
+delete_rows_main_ALT_0_Block_5.<noName>[1] <-> +"DELETE".<noName>[1];
+delete_rows_main_ALT_0_Block_5.<noName>[1] <-> +"FROM".<noName>[1];
+delete_rows_main_ALT_0_Optional_1.<noName>[1] <-> +alias.<noName>[1];
+delete_rows_main_ALT_0_Optional_2.<noName>[1] <-> +"WHERE".<noName>[1];
+delete_rows_main_ALT_0_Optional_2.<noName>[1] <-> +comparison.<noName>[1];
+alias.<noName>[1] <-> +alias_main.<noName>[1];
+alias_main.<noName>[1] <-> +IDENTIFIER.<noName>[1];
+comparison.<noName>[1] <-> +comparison_main.<noName>[1];
+comparison_main.<noName>[1] <-> +value.<noName>[1];
+comparison_main.<noName>[1] <-> +COMP_OP.<noName>[1];
+value.<noName>[1] <-> +value_main.<noName>[1];
+value_main_ALT_0.<noName>[1] <-> +column_name.<noName>[1];
+value_main_ALT_0.<noName>[1] <-> +value_main_ALT_0_Optional_1.<noName>[0..1];
+value_main_ALT_0_Optional_1.<noName>[1] <-> +".".<noName>[1];
+value_main_ALT_0_Optional_1.<noName>[1] <-> +alias.<noName>[1];
+value_main_ALT_1.<noName>[1] <-> +constant.<noName>[1];
+constant.<noName>[1] <-> +constant_main.<noName>[1];
+constant_main_ALT_0.<noName>[1] <-> +NUMBER.<noName>[1];
+constant_main_ALT_1.<noName>[1] <-> +STRING.<noName>[1];
```

```
+query.<noName>[1] <-> +query_main.<noName>[1];
+query_main.<noName>[1] <-> +query_main_ALT_0_Block_4.<noName>[1];
+query_main.<noName>[1] <-> +"FROM".<noName>[1];
+query_main.<noName>[1] <-> +query_main_ALT_0_Optional_0.<noName>[0..1];
+query_main.<noName>[1] <-> +table_list.<noName>[1];
+query_main.<noName>[1] <-> +"SELECT".<noName>[1];
+query_main.<noName>[1] <-> +query_column_list.<noName>[1];
+query_main_ALT_0_Optional_0.<noName>[1] <-> +"DISTINCT".<noName>[1];
+query_main_ALT_0_Block_4.<noName>[1] <-> +"WHERE".<noName>[1];
+query_main_ALT_0_Block_4.<noName>[1] <-> +condition.<noName>[1];
+query_column_list.<noName>[1] <-> +query_column_list_main.<noName>[1];
+query_column_list_main_ALT_0.<noName>[1] <->
                +query_column_list_main_ALT_0_Optional_0.<noName>[0..1];
+query_column_list_main_ALT_0.<noName>[1] <-> +column_name.<noName>[1];
+query_column_list_main_ALT_0.<noName>[1] <->
                +query_column_list_main_ALT_0_Arbitrary_1.<noName>[0..*];
+query_column_list_main_ALT_0_Optional_0.<noName>[1] <-> +".".<noName>[1];
+query_column_list_main_ALT_0_Optional_0.<noName>[1] <-> +alias.<noName>[1];
+query_column_list_main_ALT_0_Arbitrary_1.<noName>[1] <->
                +query_column_list_main_ALT_0_Arbitrary_1_ALT_0_Optional_0.<noName>[0..1];
+query_column_list_main_ALT_0_Arbitrary_1.<noName>[1] <-> +column_name.<noName>[1];
+query_column_list_main_ALT_0_Arbitrary_1.<noName>[1] <-> +",".<noName>[1];
+query_column_list_main_ALT_0_Arbitrary_1_ALT_0_Optional_0.<noName>[1] <-> +".".<noName>[1];
+query_column_list_main_ALT_0_Arbitrary_1_ALT_0_Optional_0.<noName>[1] <-> +alias.<noName>[1];
+query_column_list_main_ALT_1.<noName>[1] <-> +"*".<noName>[1];
+table_list.<noName>[1] <-> +table_list_main.<noName>[1];
+table_list_main.<noName>[1] <-> +table_list_main_ALT_0_Arbitrary_1.<noName>[0..*];
+table_list_main.<noName>[1] <-> +table_name.<noName>[1];
+table_list_main.<noName>[1] <-> +table_list_main_ALT_0_Optional_0.<noName>[0..1];
+table_list_main_ALT_0_Optional_0.<noName>[1] <-> +alias.<noName>[1];
+table_list_main_ALT_0_Arbitrary_1.<noName>[1] <-> +table_name.<noName>[1];
+table_list_main_ALT_0_Arbitrary_1.<noName>[1] <->
                +table_list_main_ALT_0_Arbitrary_1_ALT_0_Optional_1.<noName>[0..1];
+table_list_main_ALT_0_Arbitrary_1.<noName>[1] <-> +",".<noName>[1];
+table_list_main_ALT_0_Arbitrary_1_ALT_0_Optional_1.<noName>[1] <-> +alias.<noName>[1];
+condition.<noName>[1] <-> +condition_main.<noName>[1];
+condition_main.<noName>[1] <-> +condition_main_ALT_0_Optional_0.<noName>[0..1];
+condition_main.<noName>[1] <-> +and_term.<noName>[1];
+condition_main_ALT_0_Optional_0.<noName>[1] <-> +"OR".<noName>[1];
+condition_main_ALT_0_Optional_0.<noName>[1] <-> +condition.<noName>[1];
+and_term.<noName>[1] <-> +and_term_main.<noName>[1];
+and_term_main.<noName>[1] <-> +comparison.<noName>[1];
+and_term_main.<noName>[1] <-> +and_term_main_ALT_0_Optional_0.<noName>[0..1];
+and_term_main_ALT_0_Optional_0.<noName>[1] <-> +"AND".<noName>[1];
+and_term_main_ALT_0_Optional_0.<noName>[1] <-> +and_term.<noName>[1];
+bye.<noName>[1] <-> +bye_main.<noName>[1];
+bye_main.<noName>[1] <-> +"QUIT".<noName>[1];

<endpackage>
```