LUÍS EDUARDO DE SOUZA AMORIM

REUSABLE LANGUAGE SPECIFICATIONS

Dissertation presented to the Federal University of Viçosa, as part of demands from the Programa de Pós-Graduação em Ciência da Computação, to get the title of *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2013

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
A524r
2013

Amorim, Luís Eduardo de Souza, 1989-
Reusable language specifications / Luís Eduardo de Souza Amorim. – Viçosa, MG, 2013.
vii, 121 f. : il. (algumas color.) ; 29 cm.

Texto em inglês.
Inclui apêndices.
Orientador: Vladimir Oliveira Di Iório.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.119-121.

1. ANTLR (Linguagem de programação de computador).
2. Linguagem de programação (Computadores) - Sintaxe.
3. Compiladores (Computadores). I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 005.13

LUÍS EDUARDO DE SOUZA AMORIM

REUSABLE LANGUAGE SPECIFICATIONS

Dissertation presented to the Federal University of Viçosa, as part of demands from the Programa de Pós-Graduação em Ciência da Computação, to get the title of *Magister Scientiae*.

APPROVED: August 30, 2013.

Roberto da Silva Bigonha

Mariza Andrade da Silva Bigonha

Alcione de Paiva Oliveira
(Co-adviser)

Vladimir Oliveira Di Iorio
(Adviser)

"I'm in the war of my life, at the core of my life, got no choice but to fight 'til it's done..." (John Mayer)

Acknowledgements

First I would like to thank God, without Him pulling His strings, I would not be here for sure.

I would also like to thank my family for all the support during these 7 years away from home. They believed in me more than I believed in myself.

I would like to thank Anna, that in all these years has never gave up on me. I have learned incredible things with you. I owe you a lot.

To all my friends in Viçosa, that made this city a home to me. People from '1521', who were the first to teach me what I needed to learn back then: to grow up. People from 'República Lotação' who, even for a short period of time, were like brothers to me. I am going to keep your friendship for the rest of my life. To the 'broderz' from Computer Science, who made my life in graduation and masters more pleasant. And the people from Math graduation, for all the laughs and moments that I will never forget.

To my friends from Bonfinópolis, for being present even far away. I will always take you with me wherever I go.

To the 'online friends' that I never actually met, but showed me that it is possible to touch a person's life from distance.

To all my teachers that were way more than teachers, they became really great friends, specially André and my adviser Vladimir, who gave me the basis so I could pursue my dream and become the person I am today.

To the people from 'Sistema Financiar', who taught me that there is always something to learn.

Finally I would like to thanks Capes for the financial support.

Contents

Resumo

AMORIM, Luís Eduardo de Souza, M.Sc., Universidade Federal de Viçosa, agosto de 2013. **Especificações Reutilizáveis de Linguagens.** Orientador: Vladimir Oliveira Di Iorio. Co-orientadores: Ricardo Santos Ferreira e Alcione de Paiva Oliveira

O processo de construção de linguagens de programação não é uma tarefa fácil. Programadores gastam bastante tempo desenvolvendo novas linguagens a partir do zero. O suporte ao reuso de ferramentas para geração automática de analisadores sintáticos é essencial para este processo de desenvolvimento, contudo, grande parte das ferramentas não permitem definições totalmente reutilizáveis, uma vez que estas misturam elementos sintáticos e semânticos. Além disso, é importante que estas ferramentas ofereçam algumas facilidades, tais quais as presentes em Ambientes de Desenvolvimento Integrado (IDE, do inglês Integrated Development Environment), que auxiliam e agilizam o trabalho dos desenvolvedores de linguagens. Nessa dissertação, propomos uma abordagem que tenta separar elementos sintáticos e semânticos nas definições, e também permite aos programadores utilizar facilidades de uma IDE para construir a semântica de suas definições. Nós utilizamos essa abordagem para implementar uma solução para a ferramenta ANTLR, e validamos nossa proposta utilizando gramáticas ANTLR completas, reconstruindo-as, porém reutilizando uma mesma definição sintática.

Abstract

AMORIM, Luís Eduardo de Souza, M.Sc., Universidade Federal de Viçosa, August, 2013. **Reusable Language Specifications.** Advisor: Vladimir Oliveira Di Iorio. Co-advisors: Ricardo Santos Ferreira and Alcione de Paiva Oliveira

The process of building a programming language is not an easy task. Programmers spend a lot of time building new languages from scratch. The support for reuse from tools for automatic parser generation is essential to this development process, however most tools do not provide fully reusable definitions as they often mix syntactic and semantic elements. Also, it is important that these tools offer some features, such as the ones from Integrated Development Environments, that help and faster the work of language developers. In this dissertation, we propose an approach that attempts to separate syntactic and semantic elements in definitions, and also allows programmers to use features from an IDE to build the semantics of definitions. We use this approach to implement a solution to the ANTLR tool, and we validate it by using real ANTLR grammars, building the same grammars but reusing a same syntactic definition.

Chapter 1

Introduction

The process of building a programming language is not an easy task. First, it is necessary to take many design decisions, and after that, it is needed a lot of effort on the implementation.

There are many approaches and tools that may help the process of creating new programming languages. For example, Language Oriented Programming [27, 7] is an approach that consists of creating small domain specific languages (DSLs) and use them to solve a problem. As the DSLs are simple, the process of building them may be also simple. Tools like Meta Programming System [18], Intentional Programming [23], and XMF [5] apply the concepts of Language Oriented Programming.

In a way similar to software development, where a programmer can reuse code while he is developing new systems, it is also desirable for language developers to reuse parts of the definitions of other languages. Moreover, while programming, programmers can use many features from Integrated Development Environments (IDEs), such as Eclipse [8], Netbeans [16], among others, that help them build their programs easily. This also is desirable in the programming languages development, since it is possible to reuse definitions from other languages and also use IDEs features to accelerate the process.

Formal grammars and extensions of this model like EAG (Extended Attribute Grammars) [28] are the most important tools used for definition of programming languages [1]. They describe how a language will work, i.e., its syntactic and semantics. The main reasons why programmers reuse grammars are [20]:

- mixing together new grammars by using pieces from other grammars;
- combining complete grammars to embed one in another;
- deriving new variants of one language;

- copying and modifying semantics of a grammar, without changing the language.

But in order to reuse definitions, it is important that these definitions be "clean", i.e., they must not mix their syntactic and semantic elements, since semantic elements provide specific meaning to the grammar. However, most of the tools for automatic parser generation, such as ANTLR [19], most YACC descendents [14], JavaCC [6], SableCC [11], still do not produce a fully reusable grammar, as they mix both syntax and semantics in the grammar. Furthermore, semantic elements are often written in some programming language, and it may contain errors that will not be captured by the development environment of the tool.

One example is shown in Figure 1.1, with a small grammar to parse expressions and calculate their values, written in the ANTLR format. We can see that the grammar is polluted by Java elements that are highlighted in the text, needed to provide a specific semantics to the definition, in this case, compute its value.

```
expr returns [int result] :
    t=term {result = t;}
    ({int sign = 1;} (PLUS | MINUS {sign = -1;}) t=term {result += sign * t;})*;

term returns [int result] :
    f=factor {result = f;}
    ({boolean div = false;} (MULT | DIV {div = true;}) f=factor {
        if (div)
            result /= f;
        else
            result *= f;
     })*;

factor returns [int result] :
    n=NUMBER {result = Integer.parseInt(n);};
```

Figure 1.1: Specification in ANTLR of a grammar for expressions.

As we can see, the reuse of this grammar is difficult because its syntactic elements are bound to the semantic ones. A desirable grammar is presented in Figure 1.2. This grammar contains only syntactic elements needed to parse an expression. The semantic of the grammar can be added later, and the grammar can be reused for many purposes such as compute a value for an expression, pretty-print the expression parsed, etc.

While developing a language, it is also important that the programmer have support from the tool that he decided to use. In this case, IDEs are essential to provide features, such as code completion and error reporting, that will make the development process

```
expr : t=term ((PLUS | MINUS) t=term)* ;
term : f=factor ((MULT | DIV) f=factor)* ;
factor : n=NUMBER ;
```

Figure 1.2: Specification in ANTLR of a grammar for expressions without semantic actions.

easier and quicker. Unfortunately, most tools to build languages, such as ANTLR, do not provide features to edit definitions in both syntactic and semantic level of the grammar.

We propose a simple solution to this problem, where it would be possible to make a complete separation of syntax and semantic concerns in grammar definitions. Moreover, programmers would be able to use the features of IDE to create the semantic part of their languages. In more detail, we aim to:

- propose a methodology that allows to implement a full separation between syntactic and semantic elements that can be applied to tools for automatic parser generation, specially those whose syntax and semantics are tangled in definitions;
- present an adaptation to a specific tool. In this case, we chose ANTLR, since it fills the requirements of not providing full reuse of definitions and it is one of the most used tools for automatic parser generation.

To validate our proposal, we chose real specifications of ANTLR grammars, used in its own implementation. The results obtained show that it is possible to reuse one syntactic definition and define different sets of semantic actions separately. Thus, a syntactic definition can be reused, showing that the methodology can succeed.

This dissertation is organized as follows: Chapter 2 discusses some of the related work, i.e., some tools for automatic parser generation, and how they provide reuse of their definitions. The following chapter is about the methodology used to provide the separation of syntactic and semantic concerns of a definition. Chapter 4 discusses the implementation of our solution based on our proposed methodology. After that, the following chapter presents the validation of our implementation, discussing the results obtained with some experiments. Finally, we present our conclusion in Chapter 6, pointing out the important topics of our research and presenting some future work.

Chapter 2

Related Work

Modularity and reuse were always key concepts for the development of large systems. Being able to reuse pieces of code saves a lot of time and effort in developing software. These concepts seem to be applicable to the construction of programming languages, and became an aim to tools for automatic parser generation.

There are a lot of tools for designing and implementing languages. They differ from each other basically in the class of grammars they accept or formalism that they use, their easiness of use and their support for reuse. Traditional parser generators, such as Yacc [14], ANTLR [19], Rats! [12], SableCC [24] and many others, use grammars embedded with semantic actions making the definitions difficult to reuse and hard to maintain.

In this chapter, we present some of the most common tools and techniques for parser generation and how they attempt to provide reuse of their definitions.

2.1 Rats!

Rats![12] is a tool based on modular constructions to define languages. It allows the construction of extensible and modular grammars and is based on the formalism known as *Parsing Expression Grammars* (PEG)[10]. PEG has some advantages when compared to traditional context-free grammars, such as avoiding ambiguity. Because it uses ordered choices instead of symmetric choices, meaning that the order of the options for one production is important. PEG offers unlimited lookahead, with linear parse time guaranteed by a technique known as *packrat parsing* [9]. We can also say that a definition using PEG formalism defines how to parse a language instead of how to generate one.

One of the most important feature from Rats! is that it allows organizing the definitions in modules and attach pre-existing modules to it. For example, if a new language

uses the same syntax of Java expressions, it is possible to import the appropriate module from the Java grammar and add it to this language. However, the definitions may contain semantic elements, specified in actions in the Rats! productions, which makes reuse of the definitions more difficult. An example of a Rats! production is shown in Figure 2.1. It is possible to note that the productions that contain semantic data are already attached to a semantic purpose, making difficult the reuse of the definitions.

```
FullProduction FullProduction =
  atts:ProductionAttributes
  type:Name nt:UnqualifiedNonTerminal "=":Symbol choice:Choice ";":Symbol
    { yyValue = new FullProduction(atts.list(), type, nt, choice); }
  ;
```

Figure 2.1: An example of a Rats! production.

When developing a language using Rats!, a developer may rely on its facilities for automatic deducing productions and omit explicit semantic actions. For productions that only pass the value through or do not produce any semantic value, like lexical productions, Rats! can automatically pass the value in the production for the first case, or ignore the semantic value, for the second case. Avoiding the need of creating semantic actions makes the grammar clearer and easier to modify and more reusable. But, even using these facilities, it is often necessary to embed some semantic actions to build a full language and it may pollute a grammar mixing syntactic and semantic elements.

Another point that is worth mentioning is that the code inserted inside semantic actions may contain errors that will appear only after the parser generation. This is the problem mentioned in the Introduction, where there is not support from the IDE to edit the definition in both syntactic and semantic levels of the grammar.

2.2 SDF

SDF (*Syntax Definition Formalism*) is a formalism that provides a standard interface between lexical and context-free syntax [13]. Its underlying parsing technology is based on Generalized LR parsing [2], allowing modular definitions for syntax, since the set of GLR grammars is closed under union. It means that new definitions may be added to a GLR grammar without worrying about conflicts. Restricted classes such as LALR grammars, commonly used in tools such as YACC [14], do not present these facilities: combining two LALR grammars may not result on another LALR grammar. A payload for the flexibility

for syntax definitions of the GLR approach is a relative lack of efficiency of the parsing process.

Definitions using SDF are fully reusable, since SDF grammars only contain syntactic elements. Semantics may be expressed using the algebraic specification formalism ASF [25], defining tree matching conditional equations that work with the abstract syntax tree. A price to pay is a lack of flexibility, because the standard translation from concrete syntax to abstract syntax trees is very restrictive. For example, the order of arguments of a concrete and abstract forms of a function must always be the same. Figure 2.2 presents an example of a production written using SDF formalism. As it is shown, there are no semantic elements within a production. The *cons("Module")* is used as a constructor name that identify the production in the syntax definition. Note that left and right sides of the production are inverted if compared to regular context-free grammars.

```
"module" ID Def* -> Start {cons("Module")}
```

Figure 2.2: An example of a production written in SDF.

The tool Meta-Environment [25] is used to develop a project using the ASF+SDF formalism. It provides some features to help the programmer. The Spoofax tool [15] also uses SDF formalism, however, to define semantics, it uses Stratego [26], that is a language for software transformation based on the paradigm of rewriting strategies, that is more flexible than the Meta-Environment tool. Another big advantage of Spoofax is that it produces as result a plug-in for the Eclipse IDE [8], that allows the programmer, besides design, build a particular IDE that provides editing services to its own language.

2.3 ANTLR

ANTLR [19] is another option of tool for building languages and it is one of the most popular parser generators nowadays. ANTLR generates LL(*) recognizers [21], and for non-LL(*) grammars, backtracking gives it the power of PEG in which partial result memoization guarantees linear parse times.

A language definition written using ANTLR v3 can be divided into 3 grammars. The lexer and parser grammars are similar to the ones in other tools, being responsible to analyze the code lexically and syntactically. The novelty here is the tree grammar, that is responsible to walk through the AST generated by the parser.

Tree Grammars are used to give semantic to a definition, i.e., tree grammars can be

used to generate a translation, to extract information or to compute ancillary data. That said, it is possible to define more than one tree grammar to one language definition, for example, we may define a tree grammar that walks through an AST pretty-printing the nodes, and other to walk through the same AST, generating code. As tree grammars are generally used to perform semantic actions, they are often polluted, meaning that the definitions to walk through the AST are mixed with semantic actions, making them hard to reuse.

```
option :
        ^( ASSIGN id=ID {out(id.getText()+"=");} optionValue )
;
```

Figure 2.3: An example of a tree grammar production in ANTLR v3.

Figure 2.3 shows an example of a production of a tree grammar in ANTLR. As shown in Figure 1.1, ANTLR parser grammars may contain semantic actions, although they appear more often in the tree grammars. Mixing production rules with semantic actions is not the only challenge for the reuse of tree grammars. A same AST format may be described for different tree grammars. For example, different nodes from an AST may be visited when different tasks must be performed, such as generating code from the AST or transforming the tree into a non-finite automaton. A tree grammar that could be used by both tasks must be more general, i.e., written in a way that all nodes for the tasks can be visited.

ANTLR also does not provide IDE features to edit the code within a semantic action, meaning mainly that errors in this code will be detected only after parser generation. However, there is a plug-in for the Eclipse IDE named ANTLRWorks [3] that provides features for editing the syntactic definitions.

2.4 AspectG

As we have mentioned so far, one of the main problems for the reuse of definitions in the process of building languages, is that syntactic and semantic elements may not be totally separated. In other words, we may say that semantic elements crosscut syntactic definitions.

Aspect Oriented Programming [17] is a programming paradigm that provides ways to fix crosscutting and tangling issues. By using the ideas from aspect oriented languages,

it is possible to insert code at some points (*joinpoints*) where additional aspects can be weaved. By using AOP, we can separate syntax from semantics, which is defined in aspects that will be weaved into syntax definitions, making possible to reuse them and also to build semantics in another environment.

These ideas were implemented by the Domain Specific Aspect Language (DSAL) *AspectG* [22]. AspectG is built to work with ANTLR grammars, and uses an AOP language to define pointcuts that match joinpoints in the grammar. This way, crosscuting elements can be weaved to the definition, and implemented separately. AspectG pointcuts can be constructed to match a non-terminal of the language, or to match a substring of the semantic action. The main point of using AspectG is not to completely separate syntax from semantics, but just deal with crosscuting issues within the definition, for example, while performing two semantic activities in only one semantic action. The pointcut definition is restricted, not allowing to define a new semantic action to a production.

Figure 2.4 shows a use case of AspectG. We have three different definitions. The first is an ANTLR regular production, including semantic actions. Secondly, we have a definition of a pointcut, that shows where in the ANTLR definition the code will be weaved, in this case, all the options for the non-terminal 'command'. Finally, there is an advice definition, that specifies which pointcut will be matched to the code be weaved. Inside the advice, there are two options, the code can be weaved before or after the joinpoint in the definition.

```
//production
command :(
 LEFT   {fileio.print("x=x-1;"); fileio.print("time=time+1;");}
 |RIGHT  {fileio.print("x=x+1;"); fileio.print("time=time+1;");}
 |UP    {fileio.print("y=y-1;"); fileio.print("time=time+1;");}
 |DOWN  {fileio.print("y=y+1;"); fileio.print("time=time+1;");});

//pointcut
pointcut productions(): within(command.*);

//advice
before(): productions() {dsllinenumber++; }
```

Figure 2.4: An example of AspectG definitions.

There are at least two problems with this approach to improve reuse of ANTLR definitions. One is that it does not make a full separation of syntactic and semantic concerns, i.e, these elements are still mixed in the language definition, making hard to reuse them. The second is that the programmer needs to learn a whole new language to implement the pointcuts and advices that contain the code that will be weaved to the grammar.

2.5 Grammatic

A more precise approach that uses AOP is proposed by the tool named *Grammatic* [4]. Grammatic is a tool for grammar definition and manipulation aimed to improve modularity and reuse of grammars and related development artifacts. The structure of the language syntax is not restricted, therefore it can be of any class (LL(k), LALR, or anything else). With AOP resources, it is possible to attach metadata to the grammar, including semantic elements, for example.

In Grammatic, to define pointcuts it is necessary to use a Query language. The definitions in the Query language are structured with embedded variables so they can match a production in the grammar. When this happens, it's possible to attach some metadata to the productions. A set of rules that make the attachment is called an aspect, and many aspects (independent or not) might be assigned to a single grammar. Figure 2.5 shows a query in Grammatic, and how metadata can be attached to productions, in this case all left-recursive productions will be attached with the 'leftRecursive' attribute.

```
#Rec --> #Rec ..;
[[
      Rec {
      leftRecursive;
      };
]];
```

Figure 2.5: An example of a query in Grammatic, and a metadata attachment.

As mentioned in Section 2.4, the problem of this approach is also that the programmer needs to learn a whole new language to build separate definitions of his language. Although the definitions can be completely separate, the trade-off between reusing definitions and learning a new complex language to do it may not be worth. Another problem arises when using Grammatic aspects to define code for semantic actions. These definitions are error prone, with errors being detected only after parser generation.

2.6 Our Proposal

Compared to the proposals presented in Sections 2.4 and 2.5, our approach tries to handle the crosscuting problem by a simpler way. We propose to insert annotations on the

grammar indicating points where semantic actions may be inserted. These annotations are later replaced by code following a simple protocol that we will define in Chapter 3. The annotations are provided on the format of comments, so the original meaning of the grammar is not modified.

Our approach may provide high flexibility because semantic actions can be inserted anywhere on the rules. The pollution created when mixing annotations with grammar rules is much less significant than the one caused by semantic actions in tools like Rats!, ANTLR or Yacc. We provide an implementation associated with the ANTLR tool that does not suffer of efficiency problems like the approaches based on GLR parsers (e.g. SDF). The protocol for associating annotations with code for semantic actions is very simple, so the programmer does not need to learn a complicated AOP language such as the ones used by AspectG and Grammatic. Finally, the code for semantic actions is completely separated from the grammar and can be developed in an appropriate IDE, using all the resources provided by such environment.

Chapter 3

Methodology

As we have presented in the previous chapters, the main goal of our project is to provide a way to reuse grammar definitions, making easier the process of implementing a language. Our approach is based on the ANTLR language, and it attempts to construct "clean grammars", i.e., grammars that contain only syntactic elements that could be reused afterwards. As shown on Figure 1.1, ANTLR definitions may contain both semantic and syntactic elements that make reuse more difficult.

Our first attempt is to make the definition on Figure 1.1 more like the one presented in Figure 1.2. By doing this, a programmer could reuse this definition for different purposes, for example, pretty print an expression other than calculate a value of an expression.

```
expr : t1=term /* @a1 */ ( /* @a2 */ (PLUS | MINUS /* @a3 */) t2=term /* @a4 */)* ;

term : f1=factor /* @a1 */ ( /* @a2 */ (MULT | DIV  /* @a3 */ ) f2=factor /* @a4 */)*
;

factor : n=NUMBER /* @a1 */ ;
```

Figure 3.1: Annotated Grammar.

The separation of syntactic and semantic elements of a definition can be done by at least two different approaches. One could be by the use of aspect oriented techniques, but it would require learning a new language and building a whole new definition for it. Such approach is used by the Grammatic and AspectG tools mentioned in Chapter 2.

Another solution, that we proposed in this work, is to insert some sort of "annotations" in the grammar that would be irrelevant for the ANTLR tool and use these annotations

as anchors for semantic actions that are described elsewhere. These annotations can be comments that could be discarded in the future phases of the ANTLR generation process. Each annotation is uniquely identified by a name after the symbol '@'. An example is shown in Figure 3.1. In this case, each semantic action of the ANTLR grammar will be replaced by a respective annotation.

Now it is necessary to work on the semantics of the language. To use features from an IDE, we propose that the code for the definitions of semantic actions be implemented in regular Java classes. For this purpose, it is necessary to create a protocol that relates the annotations to their specific code within the Java classes. We define two possible protocols in the following sections.

3.1 Protocol I - Many Java classes

The protocol to insert code in the desired spots (marked with the annotations) can be defined in different ways. In the first approach, we propose that the programmer must define Java classes with some requirements, which are listed below:

- for each production there would be a class with the same name as the non-terminal in the left side of a production;
- for each annotation there might be a method with the same name of the annotation;
- the returning element of a rule must be an attribute of the associated class, named 'RESULT';
- the identifiers associated with items in a production (such as *t1* and *t2* on the production with *expr* as left side in Figure 3.1) must be unique in the production; in other words, it is not allowed to use repeated names for these identifiers in the same rule.

Following this set of rules, we would have the classes presented in Figure 3.2. The code inside a method would be weaved to its related annotation, and after that a grammar similar to the one presented in Figure 1.2 is generated. To define other semantic actions to the grammar, the programmer could insert more annotations to it, and define more methods inside the corresponding class. He could also reuse the previous annotations, or ignore the ones that are useless for the new semantic purpose.

It is important to take into consideration that the classes do not have any semantic meaning, i.e., they only work if weaved into an annotated grammar. However, the code in the classes are guaranteed syntactically correct, avoiding the need of generating a parser in order to discover, later, syntactic errors in the semantic actions. We present another option for this protocol in the next section.

```
class expr{
    int t1, t2;
    int sign;
    int RESULT;

    public void  a1 (){
        RESULT = t1 ;
    }

    public void  a2 (){
        sign = 1;
    }
    ...

    public void  a4 (){
        RESULT += sign*t2;
    }
}
```

```
class term{
    int f1, f2;
    boolean div;
    int RESULT;

    public void  a1 (){
        RESULT = f1 ;
    }

    public void  a2 (){
        div = false;
    }
    ...

    public void  a4 (){
        if (div)
          RESULT /= f2;
        else
          RESULT *= f2;
    }
}
```

```
class factor{
    int n;
    int RESULT;

    public void  a1 (){
        RESULT =
            Integer.parseInt(n);
    }

}
```

Figure 3.2: Related Classes with the Annotated Grammar.

3.2 Protocol II - One Java Class

Optionally, a different protocol to link the annotations and the code of semantic actions can be defined following the guidelines below:

- there will be only one class, that will contain methods to the productions of the grammar;
- each method will be related with one production, and its name must start with "prod_" and be followed by the name of the non-terminal of the production;
- For each annotation in the grammar, there must be a related annotation followed by Java code, that afterward, will be weaved into the grammar.
- In this case, the returning element of a rule will be defined by an annotation named '@returns' that will appear before the regular annotations.
- The identifiers associated with items in a production must also have unique names in the same rule, and will be declared as attributes of the method, along with the returning element.

By using Protocol II, a class similar to the one presented in Figure 3.3 must be defined. In this case, the code following each annotation would be weaved into the grammar at the point of the related annotation. This approach is more similar to the parser generated by the ANTLR tool, where there is only one class, and at least one method to each production.

```
class ExprGrammar{
    public void prod_expr(){
        int t1, t2;
        /* @returns */
            int result;
        /* @a1 */
            result = t1;
        /* @a2 */
            int sign = 1;
        /* @a3 */
            sign = -1;
        /* @a4 */
            result += sign*t;
}
    public void prod_term(){
            ...
    }
    public void prod_factor(){
        int n;
        /* @a1 */
            result = Integer.parseInt(n);
        /* @returns */  ...
    }
}
```

Figure 3.3: Related Class with the Annotated Grammar.

The main advantage of this protocol is that there would be only one class for the entire grammar, although annotations must be inserted also in the Java class to link the code to the grammar. It is worth to mention that each protocol has its own strength and weaknesses. The implementation might be easier or harder by using one or another, and there are many other alternatives to the rules of both protocols.

Our solution is implemented using Protocol II. Since the grammar presented in this chapter is too simple to cover all the semantic elements that may appear in a definition, we propose more rules to in order to guide programmers while giving semantics to a grammar. More complex examples are presented in Figure 3.4, where we have parametrization of non-terminals and also, optional fields that may appear in a production. The productions presented are fragments of the grammars in Appedix.

To deal with these elements, we must add the following rules to our protocol:

```
grammar_
:           ^( LEXER_GRAMMAR grammarSpec["lexer " ] )
            |^( PARSER_GRAMMAR grammarSpec["parser "] )
            |^( TREE_GRAMMAR grammarSpec["tree "] )
            |^( COMBINED_GRAMMAR grammarSpec[""] )
;

grammarSpec[String gtype]
:           ...
;

action @init {
String scope=null, name=null;
String action=null;
}
:           ...
;
```

Figure 3.4: A set of productions from the ANTLRTreePrinter grammar.

- in the method of a parametrized production, the parameters of the production will be the parameters of the related method;
- to replace a call of a parametrized non-terminal in the grammar, an annotation must be inserted in both the class and the grammar, that in the class will be represented by a call to the correspondent method of the parametrized production;
- to deal with the elements @init or @after of a production, in the methods of productions that contain these elements, there must be annotations named the same as them.

For example, the related methods for each production in Figure 3.4 have to look similar to those presented in Figure 3.5.

This approach also improves the correctness of the Java code. Making a correspondence between the parameters of a production and the parameters of the method forces the programmer to set them correctly. The translation of the method call to the parameter format in the production is done by the weaver.

There is still a problem that our protocol does not cover. Grammars in ANTLR may have extra fields that will be attached to the generated parser. These members are defined in the "@header" and "@members" fields. The *@header* defines which package will contain the parser that will be generated and extra imports used by it. The *@members* field contains extra methods and attributes of the generated parser. These elements also pollute a definition and must be extracted to improve reuse.

```
public void prod_grammar_() {
      /* @a1 */
      prod_grammarSpec("lexer");
      /* @a2 */
      prod_grammarSpec("parser");
            ...
}

public void prod_grammarSpec (String gtype){
            ...
}

public void prod_action() {
      /* @init */
      String scope=null, name=null;
      String action=null;
}
```

Figure 3.5: Methods for a parametrized production, and a production that has the '@init' field.

The following rules were defined to extract these items from our related class:

- the items within the *@header* will be the same as the ones in the related class;
- the items within the *@members* will be members of the related class, whether being methods or attributes.

The whole protocol is presented in Table 3.1, which shows, for each rule, how to deal with the semantic elements that pollute a definition. Figure 3.6 illustrates the application of the protocol, showing a general example of a grammar and its related class. Methods starting with "prod_" will not be included in the *@members* field when weaved to the annotated grammar.

With this model, the reuse is guaranteed, since there is a relationship many-to-one between the grammar and the classes. The annotated grammar can be reused with a different class each time a new semantics is required. As long as the annotations are just comments, they can be ignored if not used by the weaver. A new class can also use the previous annotations and overwrite them with its own code.

Our proposal is also flexible from the implementation point of view. The annotations in the grammar only link their name to their spot in the grammar. Also, the Java classes only link the code to the name of the annotation. A weaver can be built ignoring the way that these elements are collected, as long as they match their names. The details of our implementation are presented in the next chapter.

Grammar	Class
i) Annotated grammar	One class (to perform one semantic activity)
ii) Production	Method ("prod_" + non terminal name)
iii) Annotation	Related annotation followed by java code
iv) Return element	Annotation named '@returns' (before other annotations)
v) Each named item of the production (unique names)	Attribute of the method
vi) Parameters of a production	Parameters of the method
vii) Parameters used inside a production (replaced by an annotation)	A call for the method of the non terminal
viii) @init or @after elements	Annotations named @init or @after
ix) @header elements	The same as the related class
x) @members elements	Members of the class

Table 3.1: Protocol to build the pair <Annotated Grammar, Related Class>

Annotated Grammar

```
tree grammar grammarName;

nonterminal :
      nonterminal2 /*@annotParam */
      /* @someAnnotation */
;
nonterminal2: ... ;
```

Related Class

```
package grammarPackage;
import grammarImport;

class grammarName{
      public Object membersAttribute;
      public void membersMethod (){ javaCode; }
      public void prod_nonterminal(){
            /* @returns */
            int someVariable;
            /* @init */
            javaCode;
            /* @annotParam */
            prod_nonterminal2(someValue);
            /* @someAnnotation */
            javaCode;
      }
      public void prod_nonterminal2(int attribute){
            ...
      }
}
```

Figure 3.6: General example of applying the protocol of Table 3.1.

Chapter 4

Implementation

Using the Protocol II, defined in Chapter 3, two files are built. One is the annotated grammar, which contains only syntactic elements plus the annotations necessary to point the places where semantics will be inserted. Another is the related class, where all semantic elements are defined, so they can be weaved to the annotated grammar, generating a regular ANTLR grammar.

4.1 Particularities to Build the Related Class

There are still some requirements to build the related class which are inherent from the implementation. The first one is that the class must extend another specific class generated automatically by our program. This base class contains all tokens of the grammar as its attributes and a default empty constructor. It also extends the class TreeParser from the ANTLR package.

Also, as mentioned in the protocol, the items of productions will become attributes of the methods. These attributes must be declared with type GrammarAST. However there are some variables in the production that are not directly associated with the non terminals of the production. These variables are predefined by the tool and provide access to specific information of the production, they are translated to the generated parser into a more complex piece of code.

In a production, we can have for example the variable $start that is related to the left hand side non terminal and can not be directly translated to Java code (because of the $). So, it is necessary to perform a adaptation of this variable to the related class. Figure 4.1 shows how to use this variable in the Java code and how it will be translated

afterward to the generated grammar. In this case there is a variable named *retval* of type *atom_return*. The $start member can be accessed by using this variable, and will be adapted by the weaver to $start.

Every time a production with a non terminal X in the left hand side uses specific variables of ANTLR tool, there must be a class named X_return that extends the class TreeRuleReturnScope containing the returning elements of the production, if any.

Finally, all methods related to a production must throw an exception of the type RecognitionException.

```
public void prod_atom() throws RecognitionException{
    atom_return retval = new atom_return();
    GrammarAST rarg = new GrammarAST() ...;
    /* @init */
    out(" ");
    /* @atom1 */
    out(((GrammarAST) retval.start).toString()); // $start.toString()
    /* @atom2 */
    out("["+rarg.toString()+"]");
    ...
}
```

Figure 4.1: The use of the $start variable.

4.2 Weaving Process

With the assumption that the class and the annotated grammar were correctly built, we can focus on the weaver. The weaver must collect all the points in the grammar where there are semantic elements (probably marked with annotations), and replace them with the related code defined in the Java class. This implies that the weaving process is done in two different phases: the collection phase and the weaving phase.

To collect the necessary information and weave it to produce the new ANTLR grammar, transformations must be done in the AST of both related class and ANTLR grammar. To perform these transformations, we decided to use ANTLR and Java grammars that are available with the ANTLR tool.

There are at least two ways to build the weaver. One would be walking through the AST of the ANTLR file collecting the nodes necessary to the process of weaving. Once this is done, we must do the same with the Java file, also collecting the necessary nodes

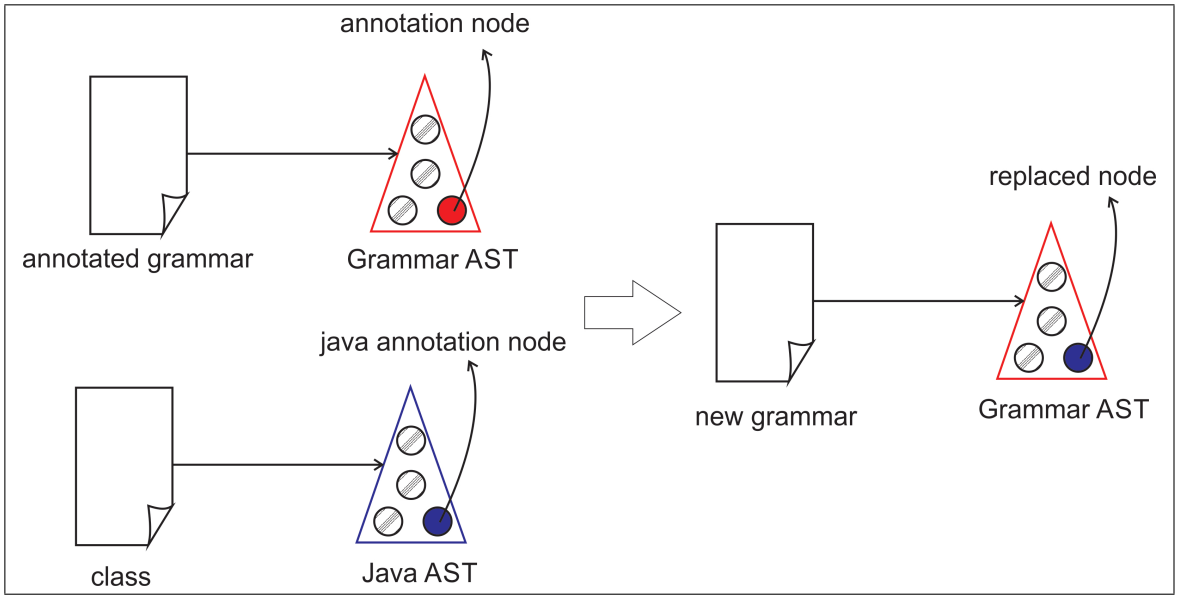

Figure 4.2: A possible implementation for the weaver.

from there. After the collection phase, the Java nodes will be weaved into the ANTLR AST, and a pretty print of this new AST will produce as result the desired ANTLR grammar. Figure 4.2 shows how this approach works.

A disadvantage of this method is that the Java and ANTLR nodes might not be compatible, so to add nodes to the ANTLR AST, a conversion between these nodes must be done. This conversion would demand knowing in advance the type of all the nodes that must be added to produce the correct result.

In the following, we discuss another approach that is simpler than the one presented above. Since the process of weaving is similar to a "copy and paste" of strings, instead of collecting Java nodes, the strings are enough if we want to paste them into the grammar, i.e., while walking through the Java AST, the strings representing the code that must be weaved will be collected to a table. Afterward, it is just a case of printing the strings in the table at the right time, while walking through the ANTLR AST.

To implement the second method, it is necessary to change the ANTLR and Java grammars. They have to identify the annotations as a different kind of comments, so in the lexer of both, rules should be added so multiple line comments starting with a '@' will be treated as an annotation. In a grammar, an annotation may appear in the same places as an ANTLR element, i.e., the option of an annotation was added to the non terminal element. In a class, an annotation may appear in the same place as a Java block.

Once the grammars are able to identify the annotations, the weaver can be finally built. In the following sections, we present the two phases necessary to build the weaver, using the second approach.

4.3 Collection Phase

This phase works together with the rules from our protocol since it defines which elements must be collected. The collection phase defines how to collect the Java code within the related class.

The same idea of changing the Java grammar were used. So, while parsing the class, it is possible to collect the necessary elements and make them available to the weaver. Following the protocol, we must first collect the elements for the *@header* field. To perform this, in the rule responsible for parsing this element, its string representation is added to a "table" that represents all elements from this field. The table is represented as a list of strings. The same procedure is done to collect all the imports from the class.

Next, we have to compose the *@members* field. The idea is the same as the one for collecting the elements for the *@header*, however, now there are methods and attributes of the class that must be collected. So we must add to the productions responsible for parsing these elements in the Java grammar, semantic actions that can add these methods and attributes to another table. It is important to note that methods whose names starts with "prod_" are not part of the *@members* field.

```
public class AnnotationContent{
    public String method;
    public String params;
    public Hashtable<String, String> annTable;

    public AnnotationContent(){
      annTable = new Hashtable<String, String>();
    }

}
```

Figure 4.3: The structure to capture all elements from a method.

Finally, we have to work with the semantic elements that may appear within a production. These elements are defined inside the "prod_" methods. To collect all these elements correctly, we defined a structure that is able to store the name of the method plus its parameters and each code inside each annotation. This structure is represented by the class shown in Figure 4.3 that has as its attributes:

- a string to store the name of the method;

- a string to store the parameters of the method;
- and a hashmap that maps the code to its annotation.

On the production that parses a method, we must add code that captures all these elements only for "prod_" methods. However, as we must capture the code for each annotation, it is also necessary to add semantic actions inside the production that recognize annotations. In this case, to decide which code will be mapped to each annotation, all code parsed while processing two annotations will belong to the first one. At the end of the method, the remaining code will belong to the last annotation parsed. After parsing the whole class, we will have a list containing the items for all the methods of the class, i.e., all the productions that have semantic elements in the grammar. Figure 4.4 shows an example of the structure generated after parsing a method.

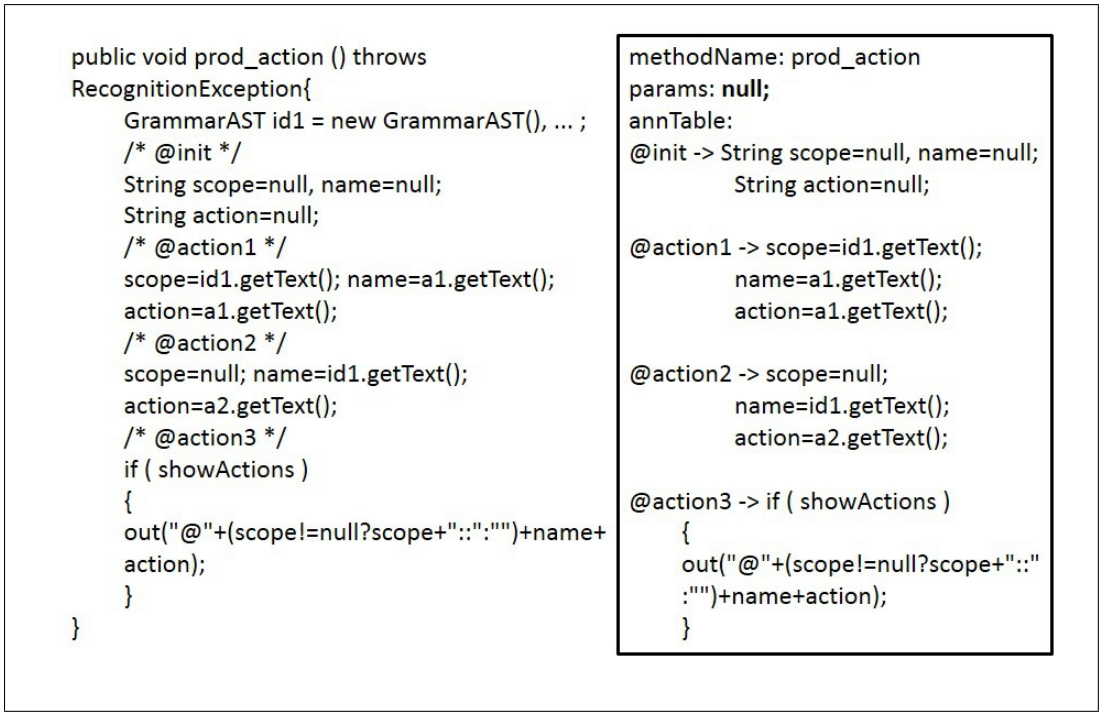

Figure 4.4: The structured created after parsing a method.

There are a few considerations that need to be made in this process of collecting these elements in the class. As mentioned before, the parameters of a production will be abstracted from it, and be described as parameters of the related method. While using a parametrized non terminal inside a production, the translation will be a call to the "prod_" method of the non terminal. One adaption of the string collected must be performed, so we can have the right one to be weaved in the grammar afterwards. In Figure 4.5, we present a method and the structure created after parsing it. Note that both translations involving parametrized non terminals must be adapted so the correct string can be stored.

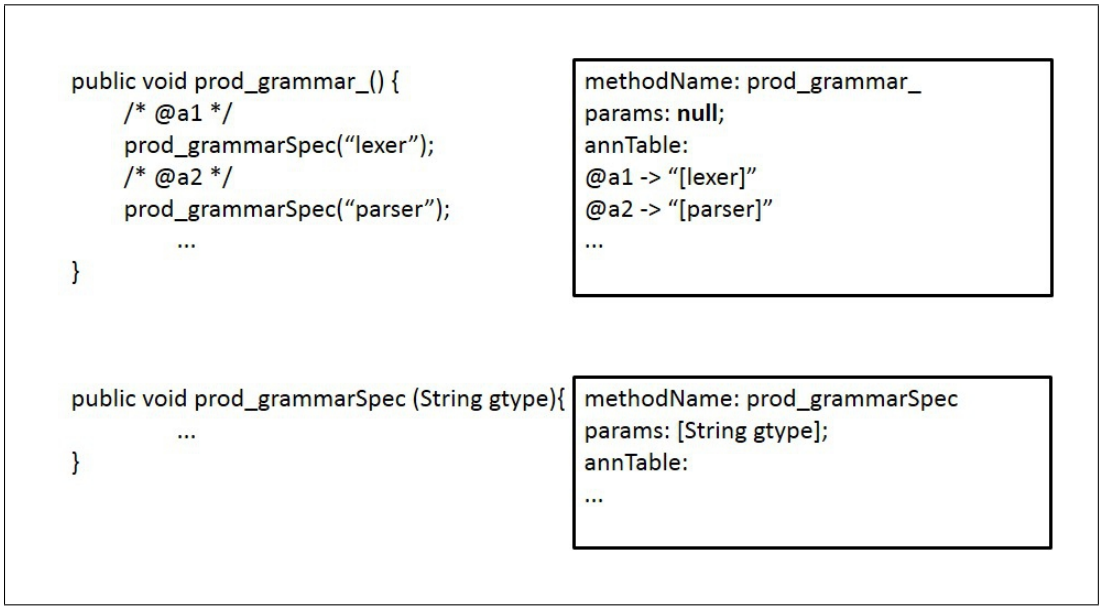

Figure 4.5: The structured created after parsing a parametrized method.

A last consideration is associated with the use of the $start attribute. We mentioned before that its use would require declaring a variable of the type TreeRuleReturnScope named retval. While collecting code, if we have a use of this specific variable, a translation must be done as shown in Figure 4.1. As show in the comment in the code of the *@atom1* annotation, the string stored for it, is $start.toString() instead of ((GrammarAST) retval).toString().

4.4 Weaving Phase

After the collection phase, the weaving phase starts. With all elements stored in the memory, in form of strings, it is necessary to decide when to print them to generate the new grammar, i.e., in which spots they must be inserted while printing the annotated grammar. To weave the strings, we decided to use a tree printer for ANTLR grammars from the ANTLR tool. Changes were added to it, so it could be able to recognize the nodes of the annotations. Beside that, it was necessary to make some more changes to it, adding new semantic actions, in order to properly produce the correct grammar at the end of the process.

While the collection phase uses the related class, the weaving phase is done with the annotated grammar. The first thing to include into the annotated grammar while generating the new grammar is the *@header* field. The annotated grammar does not

contain this field, because it was defined in the related class and collected in the previous phase. So, in the production that walks through the node that represents the *@header* element, we must add a semantic action that prints the code in the table constructed in the collection phase. By doing this, the printed grammar will contain the *@header* field as it was defined in the class.

The same procedure works for the *@members* field. In the production for walking through the node that represents this element, more code is added to perform calls of the printer for the previously collected table that contains the methods of the *@members* defined in the class.

Following the same steps of the collection phase described in Section 4.3, we have to work now with the semantic elements that may appear within a production. These elements are in the list of the structure described before that contains the name of the method (the left hand side non terminal), its parameters (the parameters of the production) and its annotations (the semantic elements of the production).

```
tree grammar grammarName;

@header{
      package grammarPackage;
      import grammarImport;
}
@members{
      public Object membersAttribute;
      public void membersMethod (){
            javaCode;
      }
}

nonterminal returns [int someVariable]
@init{javaCode;} :
      nonterminal2 [someValue]
      { javaCode;}
;
nonterminal2[int attribute] : ... ;
```

Figure 4.6: The generated grammar after weaving the two files of Figure 3.6.

In the production that walks through the production node, first, we must add its parameters, if it has any. Then, the same is done with the returning element, but it is necessary now to consult the list if there is, in the element representing a method, an annotation named *@returns* in its hashmap. This procedure can be repeated for the *@init* (or *@after*), also consulting in the hashmap if there is code associated with any of the annotations that represent these fields.

To finish the weaving phase, we must deal with the annotations. While walking through the nodes representing the annotations, it is necessary to find, in the list, the element that represent the non terminal associated with the annotation, and if there is code defined for it, to finally print the code instead of the annotation.

Figure 4.6 shows the result of applying this weaver to the pair <Annotated grammar, Related class> presented in Figure 3.6. Since the code was defined previously in the related class, it implies that all the code within the semantic actions is syntactically correct, i.e., when using the new grammar to generate a parser in the ANTLR tool, the parser will not contain syntactic errors.

4.5 Final Considerations

With the implementation described in this chapter, we have as result a functional ANTLR grammar that was built reusing a general grammar.

The whole process consists of three different phases: construction phase, collection phase and weaving phase. In the construction phase, the programmer applies the protocol to build the annotated grammar and the related class, considering the reuse of a previously built definition. The collection and weaving phases are done by our weaver. These phases can be changed later, so the approach to collect and weave can be constructed differently.

Also, the protocol to define the class and the annotations can be different, but if the structure of the collecting phase is preserved, the weaving phase can remain the same.

Figure 4.7 shows how our proposal works. The dashed line represents the work that should be done by the programmer. The other items are done by our weaver. Of course, the reuse of a definition is optional in the construction phase. Finally, after the weaving phase, the result produced is the generated grammar.

Our method works for both tree and parse grammars, but our intent is to use it to reuse specially tree grammars, since they are often more polluted with semantic elements.

Another reason to apply our methodology mainly to tree grammars is because tree grammars are more often reused in different situations, when compared to parse grammars. For example, it is very common to produce an abstract syntax tree (AST) after the parser pass of a compiler, and the following passes, such as static semantic analysis, code generation etc, may be defined as visitors for this AST. Our methodology is ideal in this case: a generic visitor for the AST may be defined as a tree grammar without semantic actions. This grammar may be reused several times, each one with a specific set of semantic actions defined using the related classes.

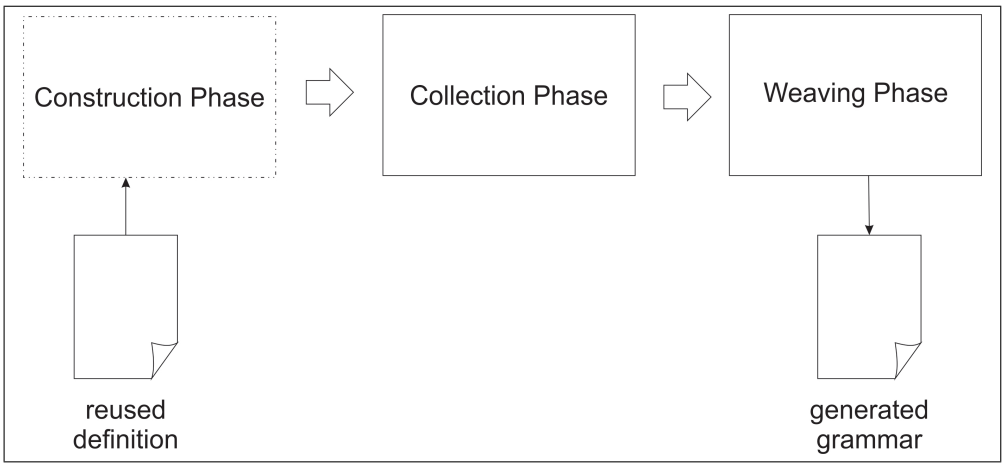

Figure 4.7: A diagram that illustrates all the phases to construct an ANTLR grammar that may reuse another grammar.

Chapter 5

Validation

As we mentioned before, ANTLR tree grammars are usually defined with the aim of walking through an AST and performing a semantic activity with it, such as generate code, pretty print, etc. For this reason, tree grammars are more often polluted with semantic actions, which make its reuse very difficult. Another reason why it is difficult to reuse tree grammars is that since they are built to a specific purpose, they may not visit all nodes of a AST, expanding only the nodes that are necessary to the semantic activity.

To validate our proposal, we decided to test it with a set of real ANTLR tree grammars that belong to the ANTLR package. But first, it was necessary to build a general tree grammar that should visit all nodes visited by each grammar of the selected set. All files discussed in this chapter are attached to this dissertation and presented as appendices.

In Appendix A, we present the three grammars used to demonstrate our proposal. The two first grammars are designed to the same language, i.e., they could be built in a way that one general grammar could be reused to build them. The other grammar is the one used as our weaver. In Chapter 4, we mentioned that the weaver is an adaption of the ANTLR Tree Printer grammar, so it can be derived from it, by reusing the general grammar with minor changes. The three grammars are: ANTLRTreePrinter.g, AssignTokenTypesWalker.g and ANTLRWeaver.g. The first is a pretty print of ANTLR grammars, the second is used to compute the token types for all literals and rules of a grammar and the third is the weaver.

The file ANTLRGeneralGrammar.g presented in Appendix B is the general grammar built to visit the nodes of both the ANTLRTreePrinter and AssignTokenTypesWalker. As we said, a tree grammar may skip or expand some nodes according to the semantic activity it is performing. An example is presented in Figure 5.1, where we have two productions of these two grammars. In ANTLRTreePrinter, the node ruleAction is expanded in the rule production, while in AssignTokenTypesWalker the node is skipped and treated differently. So in the general grammar this node needs to be expanded to cover both cases.

ANTLRTreePrinter.g

```
rule
:       ^(RULE id=ID
        (modifier)?
        ^(ARG (arg=ARG_ACTION)? )
        ^(RET (ret=ARG_ACTION)? )
        (throwsSpec)?
        (optionsSpec)?
        (ruleScopeSpec)?
        (ruleAction)*
        b=block
        (exceptionGroup)?
        EOR       )
;
```

AssignTokenTypesWalker.g

```
rulebody
:       ^(RULE id=ID
        (modifier)?
        ^(ARG (arg=ARG_ACTION)? )
        ^(RET (ret=ARG_ACTION)? )
        (throwsSpec)?
        (optionsSpec)?
        (ruleScopeSpec)?
        ( ^(AMPERSAND .*) )*
        b=block
        (exceptionGroup)?
        EOR       )
;
```

Figure 5.1: Two productions of tree grammars built for the same language that expand and skip a node, respectively.

Once we have the general grammar, the first step to the programmer is to insert annotations to it, according to the semantic elements in the original grammars. The annotations must follow Protocol II, which was defined in Chapter 3. All spots that contain semantic elements must be covered by annotations so the code can be weaved to them afterward. The general grammar with annotations covers semantic elements from both grammars.

Next, it is necessary to build the related classes based on the annotated grammar. This procedure also follows the rules defined in Protocol II. Note that the process of building the pair <Annotated Grammar, Related Class> can be done at the same time as the annotations are inserted in the general grammar. In this case, there is one class to represent the semantic elements from each grammar. These classes are presented in Appendix C.

Now that the pair is built, the only thing left is to pass them to the weaver to generate the ANTLR functional grammars. They must look similar to the ones from Appendix A, and have the same semantics.

The case of the grammar ANTLRWeaver is a little bit different since its general grammar is built to work with the ANTLR grammar that is able to recognize annotations. So it is necessary to change the tokens recognized by the grammar as well as visit the node correspondent to the annotations, but the whole general grammar stays the same. These changes in the general grammar are highlighted in Figure 5.2.

Finally, Appendix D includes the results produced by our weaver. The three generated grammars contain the same semantics as the ones in Appendix A, however, they were built by reusing the general tree grammar ANTLRGeneralGrammar. Another important point

```
options
{
 tokenVocab = ANTLRAnnotationNodes;
 ASTLabelType = GrammarAST;
}


element
 : ann=ANN_COMMENT
| ...
;
```

Figure 5.2: Changes made to the general grammar to reuse it and build the ANTLR-Weaver.

is that the parsers generated by these three grammars do not contain syntactic errors in the code, since the code was developed using an IDE that provided features to point out the errors in advance. In the following section we present some minor comments about the validation of our proposal.

5.1 Final Considerations

As discussed in this chapter, the validation can be separated in: choosing the set of grammars for testing, building a general annotated grammar, defining the related class and finally generated the result grammars. In this section we present some minor considerations about this process.

First, we noted that building a general annotated grammar is not a difficult process because, since the tree grammars are all related to one parse grammar, there are only a few different nodes skipped and expanded between them. For example, for the production from Figure 5.1, only the production presented in Figure 5.3 is added to the ANTLRTreePrinter grammar.

In the step of annotating the general grammar, it is important to mention that much work can be saved by reusing the previous annotations. For example, the annotations for the ANTLRTreePrinter and ANTLRWeaver are almost the same, they related classes were build without insert almost any annotation to the general grammar.

So it is not always needed to include a new annotation in the general grammar to mark a spot for a new semantic element from another grammar. Figure 5.4 shows the method "prod_rule" from the related classes ANTLRTreePrinter_related and AssignTo-

```
ruleAction
      :  ^(AMPERSAND id=ID a=ACTION )
;
```

Figure 5.3: Production added to the general grammar due to expand the node ruleAction.

kenTypesWalker_related. In this case, since the annotations were inserted first to the ANTLRTreePrinter grammars, the annotation *@rule6* were reused from the first grammar and it was necessary to add only the annotation *@rule8* in the general grammar.

ANTLRTreePrinter.g

```
public void prod_rule() throws RecognitionException{
      GrammarAST id = new GrammarAST(), arg = new GrammarAST(), ret = new
      GrammarAST();
      /* @rule1 */
      out(id.getText());
      /* @rule2 */
      out("["+arg.getText()+"]");
      ...
      /* @rule7 */
      prod_block(false);
}
```

AssignTokenTypesWalker.g

```
public void prod_rule() throws RecognitionException{
      GrammarAST id = new GrammarAST();
      modifier_return m = new modifier_return();
      block_return b = new block_return();
      /* @rule8 */
      currentRuleName=id.getText();
      /* @rule6 */
      trackTokenRule(id,(GrammarAST)m.start,(GrammarAST)b.start);

}
```

Figure 5.4: prod_rule methods from ANTLRTreePrinter and AssignTokenTypesWalker related classes.

Finally, as a issue due to the advantage of not reusing annotations is that the grammar may be too polluted with the annotations itself. Every new annotation inserted in the grammar lows the readability and sometimes it can be confused to the programmer to find what annotation to use to attach code to a production. Figure 5.5 shows a small production in the General Grammar from Appendix B where it was defined three annotations. Any new annotation inserted would make difficult, for example, read the syntactic

elements in the production.

```
optionsSpec
 : ^(  OPTIONS /* @optionsSpec1 */
    (option /* @optionsSpec2 */)+
    /* @optionsSpec3 */
   )
 ;
```

Figure 5.5: Production optionsSpec in the annotated grammar.

The major points experienced while testing our proposal are discussed in the next chapter, along with the conclusions of our proposal and some future work.

Chapter 6

Conclusion

In previous chapters, we presented indications that shows that most of the modern tools for automatic parser generation do not provide support for fully reusable definitions. Syntactic and semantic elements are mixed within the grammars, making them hard to reuse. We applied our proposal to the ANTLR tool, that is among the most used tools nowadays. Syntactic definitions in ANTLR can be reused, but since they are often polluted with semantic information, reuse is not effectively available.

To separate syntax from semantics, we proposed a system with annotations, where the semantic elements in the grammar are replaced by annotations in the form of ANTLR comments that do not affect the code itself. The semantics is defined in a separate file, as a regular Java class. By using regular Java classes, it is possible to take advantage of many features from IDE's, such as code completion and error highlighting.

After completely separating these elements according to Protocol II defined in Chapter 3, the annotated grammar and its related class are merged into a definition to be used in the ANTLR tool to generate the parser. There are many advantages and some disadvantages with this method. We discuss them in the following.

The main advantage of our proposal is exactly its aim, which is to reuse definitions. The method is applicable to all ANTLR grammars, although tree grammars can be reused more often as they are used to many semantic activities. One problem to reuse tree grammars is that is necessary to build one that is general enough. Some tree grammars only walk through a restricted set of nodes, expanding only the necessary and skipping the unnecessary ones. So to reuse a tree grammar, it is important that it must visit all nodes necessary to a semantic activity.

Another advantage is the use of an IDE to provide support while building semantics. Most tools for automatic parser generation based on syntactic definitions do not offer IDE's features to develop its definitions. And when there is support from IDE's, this

support does not extend to the semantics of the language. When defining semantics as Java classes, the code that will be attached to the generated parser is guaranteed syntactically correct, i.e., syntactic errors in semantic elements are detected previously to the parser generation.

A point that is worth mentioning, is that our implementation is defined in a modular way, meaning that preserving certain requirements, it can be changed later on without affecting the whole system. For example, the weaving phase can be done without annotations in the grammar, but using an AOP language, that is effective and does not pollute the language. This optional approach would lead to minor changes in the collection phase only to link the collected code to the spots defined by the AOP language.

Some problems of our proposal is that the annotations do not really clean the language, and a general grammar full of annotations can be visually polluted. Also generic grammars may not cover all tree grammars for a defined language, being necessary to change it to deal with additional nodes to properly reuse them.

A more subtle problem that we came across, when building definitions with the examples described in Chapter 5, is that the Java classes can still contain unexpected semantic errors. Figure 6.1 shows an example of the ANTLRTreePrinter_related class, where it uses the generated parser for the tree printer, identified as an semantic error (highlighted in red) by the IDE. It can be seen that this method from the *@members* field, in this case, has to use the own generated parser. Before this parser is generated, the code will contain a syntax error. It is impossible to avoid this error using a related class that defines the semantics.

```
private ANTLRTreePrinter.block_return block(GrammarAST t, boolean forceParens)
throws RecognitionException {
   ANTLRTreePrinter other = new ANTLRTreePrinter(new CommonTreeNodeStream(t));
   other.buf = buf;
   return other.block(forceParens);
}
```

Figure 6.1: A semantic error in the related class.

Another weak point in our method is that by using the related classes to define semantics, the programmer lose the advantage of using predefined variables of the tool that encapsulate more complex code, such as the $start variable from Figure 4.1. By using the related class, the code is translated directly, with no transformations.

As a final conclusion, we may say that our experiment is satisfactory as the proposal for reusing ANTLR grammars was achieved. The grammars used in Chapter 5 were built by reusing a general grammar, and according to the protocol presented in Chapter 3, related

classes were defined. The final result was the generated ANTLR grammars similar to the ones used as input.

6.1 Future Work

As future work, it is possible to use an AOP language to define the spots for semantic elements instead of using annotations. For example, in Figure 3.1, an AOP language could define pointcuts to specify the place in the grammar where the annotations are, linking these points with the annotations name. As the annotation name already link the Java code collected in the first phase of the weaver, the aspect would be responsible for the second phase, weaving the code in the right spot in the grammar. Since our implementation is modular, the related class and the collection phase would not be affected by this change.

Also, new tests can be done to improve the reliability of our protocol. If necessary, new rules can be added to it to cover more semantic elements in the grammars. For example, for more complex grammars that define scope variables, a scope class could be built inside the related class, so the scope variable would be available to all methods of it.

Our methodology can be also applicable to other tools for parser generation. Some future work can be done to adapt it to them, by improving reuse of their definitions.

Appendix A

Tree Grammars

This Appendix contains all original tree grammars that are distributed with the ANTLR tool and are used in Chapter 5.

A.1 ANTLRTreePrinter.g

```
/*
 [The "BSD license"]
 Copyright (c) 2005-2011 Terence Parr
 All rights reserved.

 Grammar conversion to ANTLR v3:
 Copyright (c) 2011 Sam Harwell
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:
 1. Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote
    products
  derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
```

```
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
    BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
    USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
    OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/** Print out a grammar (no pretty printing).
 *
 *  Terence Parr
 *  University of San Francisco
 *  August 19, 2003
 */
tree grammar ANTLRTreePrinter;

options
{
  tokenVocab = ANTLR;
  ASTLabelType = GrammarAST;
}

@header {
package org.antlr.grammar.v3;
import org.antlr.tool.*;
import java.util.StringTokenizer;
}

@members {
protected Grammar grammar;
protected boolean showActions;
protected StringBuilder buf = new StringBuilder(300);

private ANTLRTreePrinter.block_return block(GrammarAST t, boolean
   forceParens) throws RecognitionException {
    ANTLRTreePrinter other = new ANTLRTreePrinter(new
       CommonTreeNodeStream(t));
    other.buf = buf;
    return other.block(forceParens);
}

public final int countAltsForBlock(GrammarAST t) {
    int n = 0;
    for ( int i = 0; i < t.getChildCount(); i++ )
    {
        if ( t.getChild(i).getType() == ALT )
            n++;
    }
```

```
    return n;
}

public void out(String s) {
    buf.append(s);
}

@Override
public void reportError(RecognitionException ex) {
    Token token = null;
    if (ex instanceof MismatchedTokenException) {
        token = ((MismatchedTokenException)ex).token;
    } else if (ex instanceof NoViableAltException) {
        token = ((NoViableAltException)ex).token;
    }

    ErrorManager.syntaxError(
        ErrorManager.MSG_SYNTAX_ERROR,
        grammar,
        token,
        "antlr.print:␣" + ex.toString(),
        ex );
}

/** Normalize a grammar print out by removing all double spaces
 *  and trailing/beginning stuff.  FOr example, convert
 *
 *  ( A  |  B  |  C )*
 *
 *  to
 *
 *  ( A | B | C )*
 */
public static String normalize(String g) {
    StringTokenizer st = new StringTokenizer(g, "␣", false);
    StringBuffer buf = new StringBuffer();
    while ( st.hasMoreTokens() ) {
        String w = st.nextToken();
        buf.append(w);
        buf.append("␣");
    }
    return buf.toString().trim();
}
}

/** Call this to figure out how to print */
public
toString[Grammar g, boolean showActions] returns [String s=null]
@init {
  grammar = g;
  this.showActions = showActions;
}
  :  (  grammar_
```

```
    |  rule
    |  alternative
    |  element
    |  single_rewrite
    |  rewrite
    |  EOR //{s="EOR";}
    )
    {return normalize(buf.toString());}
  ;
// --------------

grammar_
  :  ^( LEXER_GRAMMAR grammarSpec["lexer " ] )
  |  ^( PARSER_GRAMMAR grammarSpec["parser "] )
  |  ^( TREE_GRAMMAR grammarSpec["tree "] )
  |  ^( COMBINED_GRAMMAR grammarSpec[""] )
  ;

attrScope
  :  ^( 'scope' ID ruleAction* ACTION )
  ;

grammarSpec[String gtype]
  :  id=ID {out(gtype+"grammar "+$id.text);}
    (cmt=DOC_COMMENT {out($cmt.text+"\n");} )?
    (optionsSpec)? {out(";\n");}
    (delegateGrammars)?
    (tokensSpec)?
    (attrScope)*
    (actions)?
    rules
  ;

actions
  :  ( action )+
  ;

action
@init {
  String scope=null, name=null;
  String action=null;
}
  :  ^(  AMPERSAND id1=ID
      (  id2=ID a1=ACTION
        {scope=$id1.text; name=$a1.text; action=$a1.text;}
      |  a2=ACTION
        {scope=null; name=$id1.text; action=$a2.text;}
      )
    )
    {
      if ( showActions )
      {
        out("@"+(scope!=null?scope+"::":"")+name+action);
```

```
      }
    }
  ;

optionsSpec
  :  ^(  OPTIONS {out("␣options␣{");}
      (option {out(";␣");})+
      {out("}␣");}
    )
  ;

option
  :  ^( ASSIGN id=ID {out($id.text+"=");} optionValue )
  ;

optionValue
  :  id=ID            {out($id.text);}
  |  s=STRING_LITERAL {out($s.text);}
  |  c=CHAR_LITERAL   {out($c.text);}
  |  i=INT            {out($i.text);}
  ;



delegateGrammars
  :  ^( 'import' ( ^(ASSIGN ID ID) | ID )+ )
  ;

tokensSpec
  :  ^(TOKENS tokenSpec*)
  ;

tokenSpec
  :  TOKEN_REF
  |  ^( ASSIGN TOKEN_REF (STRING_LITERAL|CHAR_LITERAL) )
  ;

rules
  :  ( rule | precRule )+
  ;

rule
  :  ^(  RULE id=ID
      (modifier)?
      {out($id.text);}
      ^(ARG (arg=ARG_ACTION {out("["+$arg.text+"]");} )? )
      ^(RET (ret=ARG_ACTION {out("␣returns␣["+$ret.text+"]");} )? )
      (throwsSpec)?
      (optionsSpec)?
      (ruleScopeSpec)?
      (ruleAction)*
      {out("␣:");}
      {
```

```
        if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
          out("␣");
      }
      b=block[false]
      (exceptionGroup)?
      EOR {out(";\n");}
    )
  ;

precRule
  :  ^(  PREC_RULE id=ID
      (modifier)?
      {out($id.text);}
      ^(ARG (arg=ARG_ACTION {out("["+$arg.text+"]");} )? )
      ^(RET (ret=ARG_ACTION {out("␣returns␣["+$ret.text+"]");} )? )
      (throwsSpec)?
      (optionsSpec)?
      (ruleScopeSpec)?
      (ruleAction)*
      {out("␣:");}
      {
        if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
          out("␣");
      }
      b=block[false]
      (exceptionGroup)?
      EOR {out(";\n");}
    )
  ;

ruleAction
  :  ^(AMPERSAND id=ID a=ACTION )
    {if ( showActions ) out("@"+$id.text+"{"+$a.text+"}");}
  ;

modifier
@init
{out(($start).getText()); out("␣");}
  :  'protected'
  |  'public'
  |  'private'
  |  'fragment'
  ;

throwsSpec
  :  ^('throws' ID+)
  ;

ruleScopeSpec
  :  ^( 'scope' ruleAction* (ACTION)? ( ID )* )
  ;

block[boolean forceParens]
```

```
@init
{
int numAlts = countAltsForBlock($start);
}
  :  ^(  BLOCK
      {
        if ( forceParens||numAlts>1 )
        {
          out("␣(");
        }
      }
      (optionsSpec {out("␣:");} )?
      alternative rewrite ( {out("|");} alternative rewrite )*
      EOB   {if ( forceParens||numAlts>1 ) out(")");}
     )
  ;

alternative
  :  ^( ALT element* EOA )
  ;

exceptionGroup
  :  ( exceptionHandler )+ (finallyClause)?
  |  finallyClause
  ;

exceptionHandler
  :  ^('catch' ARG_ACTION ACTION)
  ;

finallyClause
  :  ^('finally' ACTION)
  ;

rewrite
  :  ^(REWRITES single_rewrite+)
  |  REWRITES
  |
  ;

single_rewrite
  :  ^(  REWRITE {out("␣->");}
      (  SEMPRED {out("␣{"+$SEMPRED.text+"}?");}
      )?
      (  alternative
      |  rewrite_template
      |  ETC {out("...");}
      |  ACTION {out("␣{"+$ACTION.text+"}");}
      )
    )
  ;

rewrite_template
```

```
  :  ^(  TEMPLATE
      (  id=ID {out("␣"+$id.text);}
      |  ind=ACTION {out("␣({"+$ind.text+"})");}
      )
      ^(  ARGLIST
        {out("(");}
        (  ^(  ARG arg=ID {out($arg.text+"=");}
            a=ACTION   {out($a.text);}
          )
        )*
        {out(")");}
      )
      (  DOUBLE_QUOTE_STRING_LITERAL {out("␣
         "+$DOUBLE_QUOTE_STRING_LITERAL.text);}
      |  DOUBLE_ANGLE_STRING_LITERAL {out("␣
         "+$DOUBLE_ANGLE_STRING_LITERAL.text);}
      )?
    )
  ;

element
  :  ^(ROOT element) {out("^");}
  |  ^(BANG element) {out("!");}
  |  atom
  |  ^(NOT {out("~");} element)
  |  ^(RANGE atom {out("..");} atom)
  |  ^(CHAR_RANGE atom {out("..");} atom)
  |  ^(ASSIGN id=ID {out($id.text+"=");} element)
  |  ^(PLUS_ASSIGN id2=ID {out($id2.text+"+=");} element)
  |  ebnf
  |  tree_
  |  ^( SYNPRED block[true] ) {out("=>");}
  |  a=ACTION  {if ( showActions ) {out("{"); out($a.text); out("}");}}
  |  a2=FORCED_ACTION  {if ( showActions ) {out("{{"); out($a2.text);
     out("}}");}}
  |  pred=SEMPRED
    {
      if ( showActions )
      {
        out("{");
        out($pred.text);
        out("}?");
      }
      else
      {
        out("{...}?");
      }
    }
  |  spred=SYN_SEMPRED
    {
      String name = $spred.text;
      GrammarAST predAST=grammar.getSyntacticPredicate(name);
      block(predAST, true);
```

```
      out("=>");
    }
  |  ^(BACKTRACK_SEMPRED .*) // don't print anything (auto backtrack
     stuff)
  |  gpred=GATED_SEMPRED
    {
    if ( showActions ) {out("{"); out($gpred.text); out("}?␣=>");}
    else {out("{...}?␣=>");}
    }
  |  EPSILON
  ;

ebnf
  :  block[true] {out("␣");}
  |  ^( OPTIONAL block[true] ) {out("?␣");}
  |  ^( CLOSURE block[true] )  {out("*␣");}
  |  ^( POSITIVE_CLOSURE block[true] ) {out("+␣");}
  ;

tree_
  :  ^(TREE_BEGIN {out("␣^(");} element (element)* {out(")␣");} )
  ;

atom
@init
{out("␣");}
  :  (  ^(  RULE_REF    {out($start.toString());}
        (rarg=ARG_ACTION  {out("["+$rarg.toString()+"]");})?
        (ast_suffix)?
      )
    |  ^(  TOKEN_REF    {out($start.toString());}
        (targ=ARG_ACTION  {out("["+$targ.toString()+"]");} )?
        (ast_suffix)?
      )
    |  ^(  CHAR_LITERAL  {out($start.toString());}
        (ast_suffix)?
      )
    |  ^(  STRING_LITERAL  {out($start.toString());}
        (ast_suffix)?
      )
    |  ^(  WILDCARD    {out($start.toString());}
        (ast_suffix)?
      )
    )
    {out("␣");}
  |  LABEL {out("␣$"+$LABEL.text);} // used in -> rewrites
  |  ^(DOT ID {out($ID.text+".");} atom) // scope override on rule
  ;

ast_suffix
  :  ROOT {out("^");}
  |  BANG  {out("!");}
  ;
```

A.2 AssignTokenTypesWalker.g

```
/*
 [The "BSD license"]
 Copyright (c) 2005-2011 Terence Parr
 All rights reserved.

 Grammar conversion to ANTLR v3:
 Copyright (c) 2011 Sam Harwell
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:
 1. Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote
    products
  derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
    BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
    USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
    OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/** [Warning: TJP says that this is probably out of date as of
   11/19/2005,
 *   but since it's probably still useful, I'll leave in.  Don't have
    energy
 *   to update at the moment.]
 *
 *  Compute the token types for all literals and rules etc..  There are
 *  a few different cases to consider for grammar types and a few
    situations
 *  within.
 *
 *  CASE 1 : pure parser grammar
 *  a) Any reference to a token gets a token type.
```

```
 *  b) The tokens section may alias a token name to a string or char
 *
 *  CASE 2 : pure lexer grammar
 *  a) Import token vocabulary if available. Set token types for any
    new tokens
 *     to values above last imported token type
 *  b) token rule definitions get token types if not already defined
 *  c) literals do NOT get token types
 *
 *  CASE 3 : merged parser / lexer grammar
 *  a) Any char or string literal gets a token type in a parser rule
 *  b) Any reference to a token gets a token type if not referencing
 *     a fragment lexer rule
 *  c) The tokens section may alias a token name to a string or char
 *     which must add a rule to the lexer
 *  d) token rule definitions get token types if not already defined
 *  e) token rule definitions may also alias a token name to a literal.
 *     E.g., Rule 'FOR : "for";' will alias FOR to "for" in the sense
    that
 *     references to either in the parser grammar will yield the token
    type
 *
 *  What this pass does:
 *
 *  0. Collects basic info about the grammar like grammar name and type;
 *     Oh, I have go get the options in case they affect the token
    types.
 *     E.g., tokenVocab option.
 *     Imports any token vocab name/type pairs into a local hashtable.
 *  1. Finds a list of all literals and token names.
 *  2. Finds a list of all token name rule definitions;
 *     no token rules implies pure parser.
 *  3. Finds a list of all simple token rule defs of form "<NAME> :
    <literal>;"
 *     and aliases them.
 *  4. Walks token names table and assign types to any unassigned
 *  5. Walks aliases and assign types to referenced literals
 *  6. Walks literals, assigning types if untyped
 *  4. Informs the Grammar object of the type definitions such as:
 *     g.defineToken(<charliteral>, ttype);
 *     g.defineToken(<stringliteral>, ttype);
 *     g.defineToken(<tokenID>, ttype);
 *     where some of the ttype values will be the same for aliases
    tokens.
 */
tree grammar AssignTokenTypesWalker;

options
{
  tokenVocab = ANTLR;
  ASTLabelType = GrammarAST;
}
```

```
@header {
package org.antlr.grammar.v3;

import java.util.*;
import org.antlr.analysis.*;
import org.antlr.misc.*;
import org.antlr.tool.*;

import org.antlr.runtime.BitSet;
}

@members {
protected Grammar grammar;
protected String currentRuleName;

protected static GrammarAST stringAlias;
protected static GrammarAST charAlias;
protected static GrammarAST stringAlias2;
protected static GrammarAST charAlias2;

@Override
public void reportError(RecognitionException ex)
{
    Token token = null;
    if (ex instanceof MismatchedTokenException) {
        token = ((MismatchedTokenException)ex).token;
    } else if (ex instanceof NoViableAltException) {
        token = ((NoViableAltException)ex).token;
    }

    ErrorManager.syntaxError(
        ErrorManager.MSG_SYNTAX_ERROR,
        grammar,
        token,
        "assign.types:␣" + ex.toString(),
        ex);
}

protected void initASTPatterns()
{
    TreeAdaptor adaptor = new ANTLRParser.grammar_Adaptor(null);

    /*
     * stringAlias = ^(BLOCK[] ^(ALT[] STRING_LITERAL[] EOA[]) EOB[])
     */
    stringAlias = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
    {
        GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
        adaptor.addChild( alt, adaptor.create( STRING_LITERAL,
           "STRING_LITERAL" ) );
        adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
        adaptor.addChild( stringAlias, alt );
    }
```

```
    adaptor.addChild( stringAlias, adaptor.create( EOB, "EOB" ) );

    /*
     * charAlias = ^(BLOCK[] ^(ALT[] CHAR_LITERAL[] EOA[]) EOB[])
     */
    charAlias = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
    {
        GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
        adaptor.addChild( alt, adaptor.create( CHAR_LITERAL,
           "CHAR_LITERAL" ) );
        adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
        adaptor.addChild( charAlias, alt );
    }
    adaptor.addChild( charAlias, adaptor.create( EOB, "EOB" ) );

    /*
     * stringAlias2 = ^(BLOCK[] ^(ALT[] STRING_LITERAL[] ACTION[]
        EOA[]) EOB[])
     */
    stringAlias2 = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
    {
        GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
        adaptor.addChild( alt, adaptor.create( STRING_LITERAL,
           "STRING_LITERAL" ) );
        adaptor.addChild( alt, adaptor.create( ACTION, "ACTION" ) );
        adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
        adaptor.addChild( stringAlias2, alt );
    }
    adaptor.addChild( stringAlias2, adaptor.create( EOB, "EOB" ) );

    /*
     * charAlias = ^(BLOCK[] ^(ALT[] CHAR_LITERAL[] ACTION[] EOA[])
        EOB[])
     */
    charAlias2 = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
    {
        GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
        adaptor.addChild( alt, adaptor.create( CHAR_LITERAL,
           "CHAR_LITERAL" ) );
        adaptor.addChild( alt, adaptor.create( ACTION, "ACTION" ) );
        adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
        adaptor.addChild( charAlias2, alt );
    }
    adaptor.addChild( charAlias2, adaptor.create( EOB, "EOB" ) );
}

// Behavior moved to AssignTokenTypesBehavior
protected void trackString(GrammarAST t) {}
protected void trackToken( GrammarAST t ) {}
protected void trackTokenRule( GrammarAST t, GrammarAST modifier,
   GrammarAST block ) {}
protected void alias( GrammarAST t, GrammarAST s ) {}
public void defineTokens( Grammar root ) {}
```

```
protected void defineStringLiteralsFromDelegates() {}
protected void assignStringTypes( Grammar root ) {}
protected void aliasTokenIDsAndLiterals( Grammar root ) {}
protected void assignTokenIDTypes( Grammar root ) {}
protected void defineTokenNamesAndLiteralsInGrammar( Grammar root ) {}
protected void init( Grammar root ) {}
}

public
grammar_[Grammar g]
@init
{
  if ( state.backtracking == 0 )
    init($g);
}
  :  (  ^( LEXER_GRAMMAR     grammarSpec )
    |  ^( PARSER_GRAMMAR   grammarSpec )
    |  ^( TREE_GRAMMAR     grammarSpec )
    |  ^( COMBINED_GRAMMAR grammarSpec )
    )
  ;

grammarSpec
  :  id=ID
    (cmt=DOC_COMMENT)?
    (optionsSpec)?
    (delegateGrammars)?
    (tokensSpec)?
    (attrScope)*
    ( ^(AMPERSAND .*) )* // skip actions
    rules
  ;

attrScope
  :  ^( 'scope' ID ( ^(AMPERSAND .*) )* ACTION )
  ;

optionsSpec returns [Map<Object, Object> opts = new HashMap<Object,
   Object>()]
  :  ^( OPTIONS (option[$opts])+ )
  ;

option[Map<Object, Object> opts]
  :  ^( ASSIGN ID optionValue )
    {
      String key = $ID.text;
      $opts.put(key, $optionValue.value);
      // check for grammar-level option to import vocabulary
      if ( currentRuleName==null && key.equals("tokenVocab") )
      {
        grammar.importTokenVocabulary($ID,(String)$optionValue.value);
      }
    }
```

```
  ;

optionValue returns [Object value=null]
@init
{
  if ( state.backtracking == 0 )
    $value = ((GrammarAST)$start).getText();
}
  :  ID
  |  STRING_LITERAL
  |  CHAR_LITERAL
  |  INT
    {$value = Integer.parseInt($INT.text);}
  ;

delegateGrammars
  :  ^(  'import'
      (  ^(ASSIGN ID ID)
      |  ID
      )+
    )
  ;

tokensSpec
  :  ^(TOKENS tokenSpec*)
  ;

tokenSpec
  :  t=TOKEN_REF            {trackToken($t);}
  |  ^(  ASSIGN
      t2=TOKEN_REF       {trackToken($t2);}
      ( s=STRING_LITERAL {trackString($s); alias($t2,$s);}
      | c=CHAR_LITERAL   {trackString($c); alias($t2,$c);}
      )
    )
  ;

rules
  :  rule+
  ;

rule
  :  ^(RULE ruleBody)
  |  ^(PREC_RULE ruleBody)
  ;

ruleBody
  :  id=ID {currentRuleName=$id.text;}
    (m=modifier)?
    ^(ARG (ARG_ACTION)?)
    ^(RET (ARG_ACTION)?)
    (throwsSpec)?
    (optionsSpec)?
```

```
    (ruleScopeSpec)?
    ( ^(AMPERSAND .*) )*
    b=block
    (exceptionGroup)?
    EOR
    {trackTokenRule($id,$m.start,$b.start);}
  ;

modifier
  :  'protected'
  |  'public'
  |  'private'
  |  'fragment'
  ;

throwsSpec
  :  ^('throws' ID+)
  ;

ruleScopeSpec
  :  ^( 'scope' ( ^(AMPERSAND .*) )* (ACTION)? ( ID )* )
  ;

block
  :  ^(  BLOCK
      (optionsSpec)?
      ( alternative rewrite )+
      EOB
    )
  ;

alternative
  :  ^( ALT (element)+ EOA )
  ;

exceptionGroup
  :  ( exceptionHandler )+ (finallyClause)?
  |  finallyClause
  ;

exceptionHandler
  :  ^('catch' ARG_ACTION ACTION)
  ;

finallyClause
  :  ^('finally' ACTION)
  ;

rewrite
  :  ^(REWRITES ( ^(REWRITE .*) )* )
  |
  ;
```

```
element
  :  ^(ROOT element)
  |  ^(BANG element)
  |  atom
  |  ^(NOT element)
  |  ^(RANGE atom atom)
  |  ^(CHAR_RANGE atom atom)
  |  ^(ASSIGN ID element)
  |  ^(PLUS_ASSIGN ID element)
  |  ebnf
  |  tree_
  |  ^( SYNPRED block )
  |  FORCED_ACTION
  |  ACTION
  |  SEMPRED
  |  SYN_SEMPRED
  |  ^(BACKTRACK_SEMPRED .*)
  |  GATED_SEMPRED
  |  EPSILON
  ;

ebnf
  :  block
  |  ^( OPTIONAL block )
  |  ^( CLOSURE block )
  |  ^( POSITIVE_CLOSURE block )
  ;

tree_
  :  ^(TREE_BEGIN element+)
  ;

atom
  :  ^( RULE_REF (ARG_ACTION)? )
  |  ^( t=TOKEN_REF (ARG_ACTION )? ) {trackToken($t);}
  |  c=CHAR_LITERAL   {trackString($c);}
  |  s=STRING_LITERAL {trackString($s);}
  |  WILDCARD
  |  ^(DOT ID atom) // scope override on rule
  ;

ast_suffix
  :  ROOT
  |  BANG
  ;
```

A.3 ANTLRWeaver.g

```
/*
 [The "BSD license"]
```

```
 Copyright (c) 2005-2011 Terence Parr
 All rights reserved.

 Grammar conversion to ANTLR v3:
 Copyright (c) 2011 Sam Harwell
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:
 1. Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote
     products
  derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
     BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
     USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
     OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/** Print out a grammar (no pretty printing).
 *
 *  Terence Parr
 *  University of San Francisco
 *  August 19, 2003
 */
tree grammar ANTLRWeaver;

options
{
  tokenVocab = ANTLRAnnotationNodes;
  ASTLabelType = GrammarAST;
}

@header {
package org.antlr.grammar.reusabledefinitions;
import org.java.reusabledefinitions.JavaAnnotationNodesParser;
import org.antlr.tool.*;
```

```
import java.util.StringTokenizer;
}

@members {
protected Grammar grammar;
protected boolean showActions;
protected StringBuilder buf = new StringBuilder(300);
public String processingProduction = null;

private ANTLRWeaver.block_return block(GrammarAST t, boolean
   forceParens) throws RecognitionException {
    ANTLRWeaver other = new ANTLRWeaver(new CommonTreeNodeStream(t));
    other.buf = buf;
    return other.block(forceParens);
}

public final int countAltsForBlock(GrammarAST t) {
    int n = 0;
    for ( int i = 0; i < t.getChildCount(); i++ )
    {
        if ( t.getChild(i).getType() == ALT )
            n++;
    }

    return n;
}

public void out(String s) {
    buf.append(s);
}

@Override
public void reportError(RecognitionException ex) {
    Token token = null;
    if (ex instanceof MismatchedTokenException) {
        token = ((MismatchedTokenException)ex).token;
    } else if (ex instanceof NoViableAltException) {
        token = ((NoViableAltException)ex).token;
    }

    ErrorManager.syntaxError(
        ErrorManager.MSG_SYNTAX_ERROR,
        grammar,
        token,
        "antlr.print:␣" + ex.toString(),
        ex );
}

/** Normalize a grammar print out by removing all double spaces
 *  and trailing/beginning stuff.  FOr example, convert
 *
 *  ( A  |  B  |  C )*
 *
```

```
 *  to
 *
 *  ( A | B | C )*
 */
public static String normalize(String g) {
    StringTokenizer st = new StringTokenizer(g, "␣", false);
    StringBuffer buf = new StringBuffer();
    while ( st.hasMoreTokens() ) {
        String w = st.nextToken();
        buf.append(w);
        buf.append("␣");
    }
    return buf.toString().trim();
}
}

/** Call this to figure out how to print */
public
toString[Grammar g, boolean showActions] returns [String s=null]
@init {
  grammar = g;
  this.showActions = showActions;
}
  :  (  grammar_
    |  rule
    |  alternative
    |  element
    |  single_rewrite
    |  rewrite
    |  EOR //{s="EOR";}
    )
    {return normalize(buf.toString());}
  ;

// --------------

grammar_
  :  ^( LEXER_GRAMMAR grammarSpec["lexer␣" ] )
  |  ^( PARSER_GRAMMAR grammarSpec["parser␣"] )
  |  ^( TREE_GRAMMAR grammarSpec["tree␣"] )
  |  ^( COMBINED_GRAMMAR grammarSpec[""] )
  ;

attrScope
  :  ^( 'scope' ID ruleAction* ACTION )
  ;

grammarSpec[String gtype]
  :  id=ID {out(gtype+"grammar␣
     "+JavaAnnotationNodesParser.className+"_generated;\n\n");}
    (cmt=DOC_COMMENT {out($cmt.text+"\n");} )?
    (optionsSpec {out("\n");})?
    (delegateGrammars)?
```

```
    (tokensSpec)?
    (attrScope)*
    (actions)?
    rules
  ;

actions
  :  ( action )+
  ;

action
@init {
  String scope=null, name=null;
  String action=null;
}
  :  ^(  AMPERSAND id1=ID
      (  id2=ID a1=ACTION
        {scope=$id1.text; name=$a1.text; action=$a1.text;}
      |  a2=ACTION
        {scope=null; name=$id1.text; action=$a2.text;}
      )
    )
    {
      out("\n@header␣{");
      out(action);
    for (String h : JavaAnnotationNodesParser.header){
            out(h+"\n");
      }
     out("}\n\n");

    out("\n@members␣{\n");
    for (String m : JavaAnnotationNodesParser.members){
            out(m+"\n\n");
      }
    out("}\n\n");


    }
  ;

optionsSpec
  :  ^(  OPTIONS {out("␣options␣{\n");}
      (option {out(";␣\n");})+
      {out("}␣\n");}
    )
  ;

option
  :  ^( ASSIGN id=ID {out($id.text+"=");} optionValue )
  ;

optionValue
  :  id=ID            {out($id.text);}
```

```
  |  s=STRING_LITERAL {out($s.text);}
  |  c=CHAR_LITERAL   {out($c.text);}
  |  i=INT            {out($i.text);}
  ;



delegateGrammars
  :  ^( 'import' ( ^(ASSIGN ID ID) | ID )+ )
  ;

tokensSpec
  :  ^(TOKENS tokenSpec*)
  ;

tokenSpec
  :  TOKEN_REF
  |  ^( ASSIGN TOKEN_REF (STRING_LITERAL|CHAR_LITERAL) )
  ;

rules
  :  ( rule | precRule )+
  ;

rule
  :  ^(  RULE id=ID
      {processingProduction = $id.text;}
      (modifier)?
      {out($id.text);}
      ^(ARG {
        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.params != null)
              out(c.params);
            }
        }
      }(arg=ARG_ACTION
      //out("["+$arg.text+"]");}
      )? )
      ^(RET {
        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.annTable.get("/* @returns */") != null)
              out(" " + c.annTable.get("/* @returns */"));
            }
        }
      }(ret=ARG_ACTION
      //{out(" returns ["+$ret.text+"]\n");}
      )? )
      (throwsSpec)?
      (optionsSpec)?
```

```
      {

        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
              if (c.method.equals(processingProduction)){
                if (c.annTable.get("/* @options */") != null)
                  out("\noptions " + "{\n" + c.annTable.get("/*
                     @options */") + "}\n");
              }
        }
        //if ( showActions ) out("\n@"+$id.text+"{\n"+$a.text+"\n}\n");
      }
      (ruleScopeSpec)?
      (ruleAction)*
      {

        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
              if (c.method.equals(processingProduction)){
                if (c.annTable.get("/* @init */") != null)
                  out("\n@init " + "{\n" + c.annTable.get("/* @init
                     */") + "}\n");

                if (c.annTable.get("/* @after */") != null)
                  out("\n@after " + "{\n" + c.annTable.get("/* @init
                     */") + "}\n");
              }
        }
        //if ( showActions ) out("\n@"+$id.text+"{\n"+$a.text+"\n}\n");
      }
      {out(" :\n");}
      {
        if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
          out(" ");
      }
      b=block[false]
      (exceptionGroup)?
      EOR {out(";\n\n");}
    )
  ;

precRule
  :  ^(  PREC_RULE id=ID
      (modifier)?
      {out($id.text);}
      ^(ARG {
        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.params != null)
              out(c.params);
            }
        }
```

```
      }(arg=ARG_ACTION //{out("["+$arg.text+"]");}
      )? )
      ^(RET {
        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.annTable.get("/* @returns */") != null)
              out(" " + c.annTable.get("/* @returns */"));
            }
        }
      }(ret=ARG_ACTION // {out(" returns ["+$ret.text+"]\n");}
      )? )
      (throwsSpec)?
      (optionsSpec)?
      (ruleScopeSpec)?
      (ruleAction)*
      {
        out("\n@");
        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
              if (c.method.equals(processingProduction)){
                if (c.annTable.get("/* @init */") != null)
                  out("init " + "{\n" + c.annTable.get("/* @init */") +
                      "}\n");

                if (c.annTable.get("/* @after */") != null)
                  out("after " + "{\n" + c.annTable.get("/* @init */")
                      + "}\n");
            }
        }
        //if ( showActions ) out("\n@"+$id.text+"{\n"+$a.text+"\n}\n");
      }
      {out(" :\n");}
      {
        if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
          out(" ");
      }
      b=block[false]
      (exceptionGroup)?
      EOR {out(";\n\n");}
    )
  ;

ruleAction
  :  ^(AMPERSAND id=ID a=ACTION )

  ;

modifier
@init
{out($modifier.start.getText()); out(" ");}
  :  'protected'
  |  'public'
```

```
  |  'private'
  |  'fragment'
  ;

throwsSpec
  :  ^('throws' ID+)
  ;

ruleScopeSpec
  :  ^( 'scope' ruleAction* (ACTION)? ( ID )* )
  ;

block[boolean forceParens]
@init
{
int numAlts = countAltsForBlock($start);
}
  :  ^(  BLOCK
      {
        if ( forceParens||numAlts>1 )
        {
          //for ( Antlr.Runtime.Tree.Tree parent = $start.getParent();
              parent != null && parent.getType() != RULE; parent =
              parent.getParent() )
          //{
          //  if ( parent.getType() == BLOCK &&
              countAltsForBlock((GrammarAST)parent) > 1 )
          //  {
          //    out(" ");
          //    break;
          //  }
          //}
          out("␣(");
        }
      }
      (optionsSpec {out("␣:");} )?
      alternative rewrite ( {out("\n|");} alternative rewrite )*
      EOB   {if ( forceParens||numAlts>1 ) out(")");}
     )
  ;

alternative
  :  ^( ALT element* EOA )
  ;

exceptionGroup
  :  ( exceptionHandler )+ (finallyClause)?
  |  finallyClause
  ;

exceptionHandler
  :  ^('catch' ARG_ACTION ACTION)
  ;
```

```
finallyClause
  :  ^('finally' ACTION)
  ;

rewrite
  :  ^(REWRITES single_rewrite+)
  |  REWRITES
  |
  ;

single_rewrite
  :  ^(  REWRITE {out("␣->");}
      (  SEMPRED {out("␣{"+$SEMPRED.text+"}?");}
      )?
      (  alternative
      |  rewrite_template
      |  ETC {out("...");}
      |  ACTION {out("␣{"+$ACTION.text+"}");}
      )
    )
  ;

rewrite_template
  :  ^(  TEMPLATE
      (  id=ID {out("␣"+$id.text);}
      |  ind=ACTION {out("␣({"+$ind.text+"})");}
      )
      ^(  ARGLIST
        {out("(");}
        (  ^(  ARG arg=ID {out($arg.text+"=");}
            a=ACTION   {out($a.text);}
          )
        )*
        {out(")");}
      )
      (  DOUBLE_QUOTE_STRING_LITERAL {out("␣
         "+$DOUBLE_QUOTE_STRING_LITERAL.text);}
      |  DOUBLE_ANGLE_STRING_LITERAL {out("␣
         "+$DOUBLE_ANGLE_STRING_LITERAL.text);}
      )?
    )
  ;

element
  :  ^(ROOT element) {out("^");}
  |  ^(BANG element) {out("!");}
  |  atom
  |  ^(NOT {out("~");} element)
  |  ^(RANGE atom {out("..");} atom)
  |  ^(CHAR_RANGE atom {out("..");} atom)
  |  ^(ASSIGN id=ID {out($id.text+"=");} element)
  |  ^(PLUS_ASSIGN id2=ID {out($id2.text+"+=");} element)
```

```
  |  ebnf
  |  tree_
  | a=ANN_COMMENT //out($a.toString());
  {

        for (JavaAnnotationNodesParser.AnnotationContent c :
            JavaAnnotationNodesParser.table){
               if (c.method.equals(processingProduction)){
                 if (c.annTable.get($a.toString()) != null) {
                   if (c.annTable.get($a.toString()).charAt(0) == '[')
                   out(c.annTable.get($a.toString()));
                   else
                   out("{" + c.annTable.get($a.toString()) + "}\n");
                 }
             }
        }

  }
  |  ^( SYNPRED block[true] ) {out("=>");}
  |  a=ACTION  {if ( showActions ) {out("{"); out($a.text); out("}");}}
  |  a2=FORCED_ACTION  {if ( showActions ) {out("{{"); out($a2.text);
     out("}}");}}
  |  pred=SEMPRED
    {
      if ( showActions )
      {
        out("{");
        out($pred.text);
        out("}?");
      }
      else
      {
        out("{...}?");
      }
    }
  |  spred=SYN_SEMPRED
    {
      String name = $spred.text;
      GrammarAST predAST=grammar.getSyntacticPredicate(name);
      block(predAST, true);
      out("=>");
    }
  |  ^(BACKTRACK_SEMPRED .*) // don't print anything (auto backtrack
     stuff)
  |  gpred=GATED_SEMPRED
    {
    if ( showActions ) {out("{"); out($gpred.text); out("}?␣=>");}
    else {out("{...}?␣=>");}
    }
  |  EPSILON
  ;

ebnf
```

```
  :  block[true] {out("␣");}
  |  ^( OPTIONAL block[true] ) {out("?␣");}
  |  ^( CLOSURE block[true] )  {out("*␣");}
  |  ^( POSITIVE_CLOSURE block[true] ) {out("+␣");}
  ;

tree_
  :  ^(TREE_BEGIN {out("␣^(");} element (element)* {out(")␣");} )
  ;

atom
@init
{out("␣");}
  :  (  ^(  RULE_REF    {out($start.toString());}
        (rarg=ARG_ACTION  {out("["+$rarg.toString()+"]");})?
        (ast_suffix)?
      )
    |  ^(  TOKEN_REF    {out($start.toString());}
        (targ=ARG_ACTION  {out("["+$targ.toString()+"]");} )?
        (ast_suffix)?
      )
    |  ^(  CHAR_LITERAL  {out($start.toString());}
        (ast_suffix)?
      )
    |  ^(  STRING_LITERAL  {out($start.toString());}
        (ast_suffix)?
      )
    |  ^(  WILDCARD    {out($start.toString());}
        (ast_suffix)?
      )
    )
    {out("␣");}
  |  LABEL {out("␣$"+$LABEL.text);} // used in -> rewrites
  |  ^(DOT ID {out($ID.text+".");} atom) // scope override on rule
  ;

ast_suffix
  :  ROOT {out("^");}
  |  BANG  {out("!");}
  ;
```

Appendix B

Annotated General Grammar

This Appendix contains the general tree grammar, named ANTLRGeneralGrammar.g, that expand all nodes used by the tree grammars in Appendix A.

```
tree grammar ANTLRGeneralGrammar;

options
{
  tokenVocab = ANTLR;
  ASTLabelType = GrammarAST;
}

@header {
package org.antlr.grammar.v3;
import org.antlr.tool.*;
}


public
grammar_

  : ( ^( LEXER_GRAMMAR   grammarSpec /* @grammar_1 */ )
    | ^( PARSER_GRAMMAR  grammarSpec /* @grammar_2 */ )
    | ^( TREE_GRAMMAR    grammarSpec /* @grammar_3 */ )
    | ^( COMBINED_GRAMMAR grammarSpec /* @grammar_4 */ )
    )
  ;

grammarSpec
  : id=ID /* @grammarSpec1 */
    (cmt=DOC_COMMENT /* @grammarSpec2 */ )?
    (optionsSpec)? /* @grammarSpec3 */
    (delegateGrammars)?
```

```
    (tokensSpec)?
    (attrScope)*
    (actions)? //( ^(AMPERSAND .*) )* // skip actions
    rules
  ;

attrScope
  : ^( 'scope' name=ID attrScopeAction* attrs=ACTION )
  ;

attrScopeAction
  : ^(AMPERSAND id=ID a=ACTION)
  ;

actions
    : ( action )+
  ;

action

  : ^(  amp=AMPERSAND id1=ID
      ( id2=ID a1=ACTION
      /* @action1 */
        | a2=ACTION
        /* @action2 */
      )
    )
    /* @action3 */
  ;


optionsSpec
  : ^(  OPTIONS /* @optionsSpec1 */
      (option /* @optionsSpec2 */)+
      /* @optionsSpec3 */
    )
  ;

option
  : ^( ASSIGN id=ID /* @option1 */ o=optionValue ) /* @option2 */
  ;

optionValue
  : id=ID             /* @optionValue1 */
  | s=STRING_LITERAL /* @optionValue2 */
  | c=CHAR_LITERAL    /* @optionValue3 */
  | i=INT            /* @optionValue4 */

  ;

delegateGrammars
  : ^( 'import' ( ^(ASSIGN ID ID) | ID )+ )
  ;
```

```
tokensSpec
  : ^(TOKENS tokenSpec*)
  ;

tokenSpec
  : t=TOKEN_REF /* @tokenSpec1 */
  | ^( ASSIGN t2=TOKEN_REF /* @tokenSpec2 */ (s=STRING_LITERAL /*
     @tokenSpec3 */
  | c=CHAR_LITERAL /* @tokenSpec4 */) )
  ;

rules
  : ( rule | precRule )+
  ;

rule
  : ^(  ru=RULE id=ID /* @rule8 */
      (m=modifier)?
      /* @rule1 */
      ^(ARG /* @rule10 */ (arg=ARG_ACTION /* @rule2 */ )? )
      ^(RET /* @rule11 */ (ret=ARG_ACTION /* @rule3 */ )? )
      (t=throwsSpec)?
      (optionsSpec)?
      /* @rule9 */
      (ruleScopeSpec /* @rule10 */)?
      (ruleAction /* @rule11 */)*
      /* @rule4 */
      /* @rule5 */
      b=block /* @rule7 */
      (exceptionGroup)?
      EOR /* @rule6 */
    )
  ;

precRule
  : ^(  PREC_RULE id=ID /* @precRule8 */
      (m=modifier)?
      /* @precRule1 */
      ^(ARG /* @precRule10 */ (arg=ARG_ACTION /* @precRule2 */ )? )
      ^(RET /* @precRule11 */ (ret=ARG_ACTION /* @precRule3 */ )? )
      (throwsSpec)?
      (optionsSpec)?
      /* @precRule9 */
      (ruleScopeSpec /* @precRule10 */)?
      (ruleAction /* @precRule11 */)*
      /* @precRule4 */
      /* @precRule5 */
      b=block /* @precRule7 */
      (exceptionGroup)?
      EOR /* @precRule6 */
    )
  ;
```

```
ruleAction
  : ^(amp=AMPERSAND id=ID a=ACTION )
      /* @ruleAction1 */
  ;

modifier
  : 'protected'
  | 'public'
  | 'private'
  | 'fragment'
  ;

throwsSpec
  : ^('throws' (id=ID )+)
  ;

ruleScopeSpec
  : ^( 'scope' attrScopeAction* (attrs=ACTION )? ( uses=ID )* )
  ;

block
  : ^(  BLOCK
      /* @block1 */
      (optionsSpec /* @block2 */)?
      ( alternative rewrite /* @block3 */)+
      EOB /* @block4 */
    )
  ;

alternative
  : ^( ALT element* EOA )
  ;

exceptionGroup
  : ( exceptionHandler )+ (finallyClause)?
  | finallyClause
  ;

exceptionHandler
  : ^('catch' ARG_ACTION a=ACTION)
  ;

finallyClause
  : ^('finally' a=ACTION)
  ;

rewrite
  : ^(REWRITES single_rewrite+)
  | REWRITES
  |
  ;
```

```
single_rewrite
  : ^(  REWRITE /* @singleRewrite1 */
      ( pred=SEMPRED  /* @singleRewrite2 */ )?
      ( rewrite_alternative
      |  rewrite_template
      |   ETC /* @singleRewrite3 */
      | act=ACTION /* @singleRewrite4 */
      )

    )
  ;

  rewrite_alternative
  :  ^( a=ALT ( ( rewrite_element )+ | EPSILON ) EOA )

  ;

  rewrite_element
  : rewrite_atom
  | rewrite_ebnf
  | rewrite_tree
  ;

  rewrite_ebnf
  : ^( OPTIONAL rewrite_block )
  | ^( CLOSURE rewrite_block )
  | ^( POSITIVE_CLOSURE rewrite_block )
  ;

  rewrite_block
  :   ^( BLOCK (rewrite_alternative | rewrite_template | ETC ) EOB ) /*
     @rewriteBlock1 */



  ;


  rewrite_tree
  :   ^(  TREE_BEGIN rewrite_atom ( rewrite_element )* )
  ;

 rewrite_atom
  : RULE_REF
  | ( ^(  TOKEN_REF
        ( a1=ARG_ACTION
        )?
      )
    | CHAR_LITERAL
    | STRING_LITERAL
    )
  | LABEL
  | a2=ACTION
  ;
```

```
rewrite_template
  : ^(  TEMPLATE
      ( id=ID /* @rewriteTemplate1 */
      | ind=ACTION /* @rewriteTemplate2 */
      )
      ^(  ARGLIST
        /* @rewriteTemplate3 */
        ( ^(  ARG arg=ID /* @rewriteTemplate4 */
            a=ACTION   /* @rewriteTemplate5 */
          )
        )*
        /* @rewriteTemplate6 */
      )
      ( dq=DOUBLE_QUOTE_STRING_LITERAL /* @rewriteTemplate7 */
      | da=DOUBLE_ANGLE_STRING_LITERAL /* @rewriteTemplate8 */
      )?
    )
  ;

element
  : ^(ROOT element) /* @element1 */
  | ^(BANG element) /* @element2 */
  | atom /* @element15 */
  | ^(NOT /* @element3 */ element)
  | ^(RANGE atom /* @element4 */ atom/)
  | ^(CHAR_RANGE atom /* @element5 */ atom )
  | ^(ASSIGN id=ID /* @element6 */ el=element)
  | ^(PLUS_ASSIGN id2=ID /* @element7 */ e2=element)
  | ebnf
  | tree_
  | ^( SYNPRED block /* @element14 */ ) /* @element8 */
  | a=ACTION  /* @element9 */
  | a2=FORCED_ACTION  /* @element10 */
  | pred=SEMPRED
  /* @element11 */
  | spred=SYN_SEMPRED
  /* @element12 */
  | ^(BACKTRACK_SEMPRED .*) // don't print anything (auto backtrack
      stuff)
  | gpred=GATED_SEMPRED
  /* @element13 */
  | EPSILON
  ;

ebnf
  :
  (dotLoop) => dotLoop // .* or .+
  | block /* @ebnf5 */ /* @ebnf1 */
  | ^( OPTIONAL block /* @ebnf6 */ ) /* @ebnf2 */
  | ^( CLOSURE block /* @ebnf7 */ ) /* @ebnf3 */
  | ^( POSITIVE_CLOSURE block /* @ebnf8 */ ) /* @ebnf4 */
```

```
  ;

dotLoop
  : ( ^( CLOSURE dotBlock )
    | ^( POSITIVE_CLOSURE dotBlock )
    )
    ;

dotBlock
  : ^( BLOCK ^( ALT WILDCARD EOA ) EOB )
  ;

tree_
  : ^(TREE_BEGIN /* @tree1 */ element (element)* /* @tree2 */ )
  ;

atom

  : ( ^(  rr=RULE_REF     /* @atom1 */
        (rarg=ARG_ACTION  /* @atom2 */)?
        (ast_suffix)?
      )
    | ^(  t=TOKEN_REF    /* @atom3 */
        (targ=ARG_ACTION /* @atom4 */ )?
        (ast_suffix)?
      ) /* @atom11 */
    | ^(  c=CHAR_LITERAL  /* @atom5 */
        (ast_suffix)?
      )
    | ^(  s=STRING_LITERAL  /* @atom6 */
        (ast_suffix)?
      )
    | ^(  WILDCARD     /* @atom7 */
        (ast_suffix)?
      )
    )
    /* @atom8 */
  | l=LABEL /* @atom9 */ // used in -> rewrites
  | ^(DOT id=ID  /* @atom10 */ atom ) // scope override on rule
  ;

  ast_suffix
  : ROOT /* @ast_suffix1 */
  | BANG  /* @ast_suffix2 */
  ;
```

Appendix C

Related Classes

This appendix contains all three related classes built after applying the protocol from Chapter 4 to the grammars from Appendix A.

C.1 ANTLRTreePrinter_related.java

```
import org.antlr.tool.*;
import java.util.StringTokenizer;

import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import java.util.*;

public class ANTLRTreePrinter_related extends
    ANTLRTreePrinter_relatedBase {

  protected Grammar grammar;
  protected boolean showActions;
  protected StringBuilder buf = new StringBuilder(300);

  private ANTLRTreePrinter.block_return block(GrammarAST t, boolean
      forceParens) throws RecognitionException {
      ANTLRTreePrinter other = new ANTLRTreePrinter(new
          CommonTreeNodeStream(t));
      other.buf = buf;
      return other.block(forceParens);
  }

  public final int countAltsForBlock(GrammarAST t) {
      int n = 0;
      for ( int i = 0; i < t.getChildCount(); i++ )
```

```
    {
        if ( t.getChild(i).getType() == ALT )
            n++;
    }

    return n;
}

public void out(String s) {
    buf.append(s);
}

@Override
public void reportError(RecognitionException ex) {
    Token token = null;
    if (ex instanceof MismatchedTokenException) {
        token = ((MismatchedTokenException)ex).token;
    } else if (ex instanceof NoViableAltException) {
        token = ((NoViableAltException)ex).token;
    }

    ErrorManager.syntaxError(
        ErrorManager.MSG_SYNTAX_ERROR,
        grammar,
        token,
        "antlr.print:␣" + ex.toString(),
        ex );
}

/** Normalize a grammar print out by removing all double spaces
 *  and trailing/beginning stuff.  FOr example, convert
 *
 *  ( A  |  B  |  C )*
 *
 *  to
 *
 *  ( A | B | C )*
 */
public static String normalize(String g) {

    StringTokenizer st = new StringTokenizer(g, "␣", false);
    StringBuffer buf = new StringBuffer();
    while ( st.hasMoreTokens() ) {
        String w = st.nextToken();
        buf.append(w);
        buf.append("␣");
    }
    return buf.toString().trim();
}
```

```
public String prod_toString(Grammar g, Boolean showActions) throws
    RecognitionException{
  /* @returns */
  String s=null;
  /* @init */
  grammar = g;
  this.showActions = showActions;
  /* @toString1 */
  return normalize(buf.toString());
}

public void prod_grammar_() throws RecognitionException{
  /* @returns */
  String s=null;
  /* @grammar_1 */
  prod_grammarSpec("lexer");
  /* @grammar_2 */
  prod_grammarSpec("parser␣");
  /* @grammar_3 */
  prod_grammarSpec("tree␣");
  /* @grammar_4 */
  prod_grammarSpec("");
}

public void prod_grammarSpec (String gtype) throws
    RecognitionException{
  GrammarAST id = new GrammarAST(), cmt = new GrammarAST();
  /* @grammarSpec1 */
  out(gtype+"grammar␣"+id.getText());
  /* @grammarSpec2 */
  out(cmt.getText()+"\n");
  /* @grammarSpec3 */
  out(";\n");
}

public void prod_action () throws RecognitionException{
  GrammarAST id1 = new GrammarAST(), a1 = new GrammarAST(), a2 = new
      GrammarAST();
  /* @init */
  String scope=null, name=null;
  String action=null;
  /* @action1 */
  scope=id1.getText(); name=a1.getText(); action=a1.getText();
  /* @action2 */
  scope=null; name=id1.getText(); action=a2.getText();
  /* @action3 */
  if ( showActions )
  {
    out("@"+(scope!=null?scope+"::":"")+name+action);
  }
}

public void prod_optionsSpec() throws RecognitionException{
```

```
    /* @optionsSpec1 */
    out("␣options␣{");
    /* @optionsSpec2 */
    out(";␣");
    /* @optionsSpec3 */
    out("}␣");
  }

  public void prod_option() throws RecognitionException{
    GrammarAST id = new GrammarAST();
    /* @option1 */
    out(id.getText()+"=");
  }

  public void prod_optionValue() throws RecognitionException{
    GrammarAST id = new GrammarAST(), s = new GrammarAST(), c = new
        GrammarAST(), i = new GrammarAST();
    /* @optionValue1 */
    out(id.getText());
    /* @optionValue2 */
    out(s.getText());
    /* @optionValue3 */
    out(c.getText());
    /* @optionValue4 */
    out(i.getText());
  }

  public void prod_rule() throws RecognitionException{
    GrammarAST id = new GrammarAST(), arg = new GrammarAST(), ret = new
        GrammarAST();
    /* @rule1 */
    out(id.getText());
    /* @rule2 */
    out("["+arg.getText()+"]");
    /* @rule3 */
    out("␣returns␣["+ret.getText()+"]");
    /* @rule4 */
    out("␣:");
    /* @rule5 */
    if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
      out("␣");
    /* @rule6 */
    out(";\n");
    /* @rule7 */
    prod_block(false);
  }

  public void prod_precRule() throws RecognitionException{
    GrammarAST id = new GrammarAST(), arg = new GrammarAST(), ret = new
        GrammarAST();
    /* @precRule1 */
    out(id.getText());
    /* @precRule2 */
```

```
    out("["+arg.getText()+"]");
    /* @precRule3 */
    out("␣returns␣["+ret.getText()+"]");
    /* @precRule4 */
    out("␣:");
    /* @precRule5 */
    if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
      out("␣");
    /* @precRule6 */
    out(";\n");
    /* @precRule7 */
    prod_block(false);

  }

  public void prod_ruleAction() throws RecognitionException{
    GrammarAST id = new GrammarAST(), a = new GrammarAST();
    /* @ruleAction1 */
    if ( showActions ) out("@"+id.getText()+"{"+a.getText()+"}");

  }

  public void prod_modifier() throws RecognitionException{
    TreeRuleReturnScope retval = new TreeRuleReturnScope();
    /* @init */
    out(((GrammarAST) retval.start).getText()); out("␣");
  }

  public void prod_block(Boolean forceParens) throws
     RecognitionException{
    TreeRuleReturnScope retval = new TreeRuleReturnScope();
    /* @init */
    int numAlts = countAltsForBlock((GrammarAST) retval.start);
    /* @block1 */
    if ( forceParens||numAlts>1 )
    {
      out("␣(");
    }
    /* @block2 */
    out("␣:");
    /* @block3 */
    out("|");
    /* @block4 */
    if ( forceParens||numAlts>1 ) out(")");
  }

  public void prod_singleRewrite() throws RecognitionException{
    GrammarAST s = new GrammarAST(), a = new GrammarAST();
    /* @singleRewrite1 */
    out("␣->");
    /* @singleRewrite2 */
    out("␣{"+s.getText()+"}?");
    /* @singleRewrite3 */
```

```
    out("...");
    /* @singleRewrite4 */
    out("␣{"+a.getText()+"}");
  }

  public void prod_rewriteTemplate() throws RecognitionException{
    GrammarAST id = new GrammarAST(), ind = new GrammarAST(), arg = new
       GrammarAST(),
        a = new GrammarAST(), dq = new GrammarAST(), da = new
           GrammarAST() ;
    /* @rewriteTemplate1 */
    out("␣"+id.getText());
    /* @rewriteTemplate2 */
    out("␣({"+ind.getText()+"})");
    /* @rewriteTemplate3 */
    out("(");
    /* @rewriteTemplate4 */
    out(arg.getText()+"=");
    /* @rewriteTemplate5 */
    out(a.getText());
    /* @rewriteTemplate6 */
    out(")");
    /* @rewriteTemplate7 */
    out("␣"+dq.getText());
    /* @rewriteTemplate8 */
    out("␣"+da.getText());
  }

  public void prod_element() throws RecognitionException{
    TreeRuleReturnScope retval = new TreeRuleReturnScope();
    GrammarAST id = new GrammarAST(), id2 = new GrammarAST(), a = new
       GrammarAST(),
        a2 = new GrammarAST(), pred = new GrammarAST(), spred = new
           GrammarAST(),
        gpred = new GrammarAST();
    /* @element1 */
    out("^");
    /* @element2 */
    out("!");
    /* @element3 */
    out("~");
    /* @element4 */
    out("..");
    /* @element5 */
    out("..");
    /* @element6 */
    out(id.getText()+"=");
    /* @element7 */
    out(id2.getText()+"+=");
    /* @element8 */
    out("=>");
    /* @element9 */
    out("{"); out(a.getText()); out("}");
```

```
    /* @element10 */
    out("{{"); out(a2.getText()); out("}}");
    /* @element11 */
    if ( showActions )
    {
      out("{");
      out(pred.getText());
      out("}?");
    }
    else
    {
      out("{...}?");
    }
    /* @element12 */
    String name = spred.getText();
    GrammarAST predAST=grammar.getSyntacticPredicate(name);
    block(predAST, true);
    out("=>");
    /* @element13 */
    if ( showActions ) {out("{"); out(gpred.getText()); out("}?␣=>");}
    else {out("{...}?␣=>");}
    /* @element14 */
    prod_block(true);

  }

  public void prod_ebnf() throws RecognitionException{
    /* @ebnf1 */
    out("␣");
    /* @ebnf2 */
    out("?␣");
    /* @ebnf3 */
    out("*␣");
    /* @ebnf4 */
    out("+␣");
    /* @ebnf5 */
    prod_block(true);
    /* @ebnf6 */
    prod_block(true);
    /* @ebnf7 */
    prod_block(true);
    /* @ebnf8 */
    prod_block(true);

  }

  public void prod_tree() throws RecognitionException{
    /* @tree1 */
    out("␣^(");
    /* @tree2 */
    out(")␣");
  }
```

```
  public void prod_atom() throws RecognitionException{
    TreeRuleReturnScope retval = new TreeRuleReturnScope();
    GrammarAST rarg = new GrammarAST(), targ = new GrammarAST(), l =
       new GrammarAST(),
        id = new GrammarAST();
    /* @init */
    out("␣");
    /* @atom1 */
    out(((GrammarAST) retval.start).toString());
    /* @atom2 */
    out("["+rarg.toString()+"]");
    /* @atom3 */
    out(((GrammarAST) retval.start).toString());
    /* @atom4 */
    out("["+targ.toString()+"]");
    /* @atom5 */
    out(((GrammarAST) retval.start).toString());
    /* @atom6 */
    out(((GrammarAST) retval.start).toString());
    /* @atom7 */
    out(((GrammarAST) retval.start).toString());
    /* @atom8 */
    out("␣");
    /* @atom9 */
    out("␣$"+l.getText());
    /* @atom10 */
    out(id.getText()+".");
  }

  public void prod_ast_suffix() throws RecognitionException{
    /* @ast_suffix1 */
    out("^");
    /* @ast_suffix2 */
    out("!");
  }
}
```

C.2 AssignTokenTypesWalker_related.java

```
import java.util.*;
import org.antlr.analysis.*;
import org.antlr.misc.*;
import org.antlr.tool.*;

import org.antlr.runtime.BitSet;


import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
```

```
import java.util.Stack;
import java.util.List;
import java.util.ArrayList;

public class AssignTokenTypesWalker_related extends
   AssignTokenTypesWalker_relatedBase {
  protected Grammar grammar;
  protected String currentRuleName;

  protected static GrammarAST stringAlias;
  protected static GrammarAST charAlias;
  protected static GrammarAST stringAlias2;
  protected static GrammarAST charAlias2;

  @Override
  public void reportError(RecognitionException ex)
  {
      Token token = null;
      if (ex instanceof MismatchedTokenException) {
          token = ((MismatchedTokenException)ex).token;
      } else if (ex instanceof NoViableAltException) {
          token = ((NoViableAltException)ex).token;
      }

      ErrorManager.syntaxError(
          ErrorManager.MSG_SYNTAX_ERROR,
          grammar,
          token,
          "assign.types:␣" + ex.toString(),
          ex);
  }

  protected void initASTPatterns()
  {
      TreeAdaptor adaptor = new ANTLRParser.grammar_Adaptor(null);

      /*
       * stringAlias = ^(BLOCK[] ^(ALT[] STRING_LITERAL[] EOA[]) EOB[])
       */
      stringAlias = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
      {
          GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
          adaptor.addChild( alt, adaptor.create( STRING_LITERAL,
             "STRING_LITERAL" ) );
          adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
          adaptor.addChild( stringAlias, alt );
      }
      adaptor.addChild( stringAlias, adaptor.create( EOB, "EOB" ) );

      /*
       * charAlias = ^(BLOCK[] ^(ALT[] CHAR_LITERAL[] EOA[]) EOB[])
       */
      charAlias = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
```

```
        {
            GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
            adaptor.addChild( alt, adaptor.create( CHAR_LITERAL,
               "CHAR_LITERAL" ) );
            adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
            adaptor.addChild( charAlias, alt );
        }
        adaptor.addChild( charAlias, adaptor.create( EOB, "EOB" ) );

        /*
         * stringAlias2 = ^(BLOCK[] ^(ALT[] STRING_LITERAL[] ACTION[]
            EOA[]) EOB[])
         */
        stringAlias2 = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
        {
            GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
            adaptor.addChild( alt, adaptor.create( STRING_LITERAL,
               "STRING_LITERAL" ) );
            adaptor.addChild( alt, adaptor.create( ACTION, "ACTION" ) );
            adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
            adaptor.addChild( stringAlias2, alt );
        }
        adaptor.addChild( stringAlias2, adaptor.create( EOB, "EOB" ) );

        /*
         * charAlias = ^(BLOCK[] ^(ALT[] CHAR_LITERAL[] ACTION[] EOA[])
            EOB[])
         */
        charAlias2 = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
        {
            GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
            adaptor.addChild( alt, adaptor.create( CHAR_LITERAL,
               "CHAR_LITERAL" ) );
            adaptor.addChild( alt, adaptor.create( ACTION, "ACTION" ) );
            adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
            adaptor.addChild( charAlias2, alt );
        }
        adaptor.addChild( charAlias2, adaptor.create( EOB, "EOB" ) );
    }

    // Behavior moved to AssignTokenTypesBehavior
    protected void trackString(GrammarAST t) {}
    protected void trackToken( GrammarAST t ) {}
    protected void trackTokenRule( GrammarAST t, GrammarAST modifier,
       GrammarAST block ) {}
    protected void alias( GrammarAST t, GrammarAST s ) {}
    public void defineTokens( Grammar root ) {}
    protected void defineStringLiteralsFromDelegates() {}
    protected void assignStringTypes( Grammar root ) {}
    protected void aliasTokenIDsAndLiterals( Grammar root ) {}
    protected void assignTokenIDTypes( Grammar root ) {}
    protected void defineTokenNamesAndLiteralsInGrammar( Grammar root ) {}
    protected void init( Grammar root ) {}
```

```
 public static class optionValue_return extends TreeRuleReturnScope {
        public Object value=null;
    };

 public static class modifier_return extends TreeRuleReturnScope {
    };

 public static class block_return extends TreeRuleReturnScope {
    };

public void prod_grammar_ (Grammar g) throws RecognitionException{
  /* @init */
  if ( state.backtracking == 0 )
    init(g);
}

public void prod_optionsSpec() throws RecognitionException {
  /* @returns */
  Map<Object, Object> opts = new HashMap<Object, Object>();
  /* @optionsSpec2 */
  prod_option(opts);
}

public void prod_option(Map<Object, Object> opts) throws
   RecognitionException {
  GrammarAST id = new GrammarAST();
  optionValue_return o = new optionValue_return();
  /* @option2 */
  String key = id.getText();
  opts.put(key, o.value);
  // check for grammar-level option to import vocabulary
  if ( currentRuleName==null && key.equals("tokenVocab") )
  {
    grammar.importTokenVocabulary(id,(String)o.value);
  }
}

public void prod_optionValue() throws RecognitionException{
  optionValue_return retval = new optionValue_return();
  GrammarAST i = new GrammarAST();
  /* @returns */
  Object value=null;
  /* @init */
  if ( state.backtracking == 0 )
    retval.value = ((GrammarAST)retval.start).getText();
  /* @optionValue4 */
  retval.value = Integer.parseInt(i.getText());
}

public void prod_tokenSpec() throws RecognitionException{
  GrammarAST t = new GrammarAST(), t2 = new GrammarAST(), s = new
     GrammarAST(), c = new GrammarAST();
```

```
    /* @tokenSpec1 */
    trackToken(t);
    /* @tokenSpec2 */
    trackToken(t2);
    /* @tokenSpec3 */
    trackString(s); alias(t2,s);
    /* @tokenSpec4 */
    trackString(c); alias(t2,c);
  }

  public void prod_rule(){
    GrammarAST id = new GrammarAST();
    modifier_return m = new modifier_return();
    block_return b = new block_return();
    /* @rule8 */
    currentRuleName=id.getText();
    /* @rule6 */
    trackTokenRule(id,(GrammarAST)m.start,(GrammarAST)b.start);
  }

  public void prod_precRule(){
    GrammarAST id = new GrammarAST();
    modifier_return m = new modifier_return();
    block_return b = new block_return();
    /* @precRule8 */
    currentRuleName=id.getText();
    /* @precRule6 */
    trackTokenRule(id,(GrammarAST)m.start,(GrammarAST)b.start);
  }

  public void prod_atom(){
    GrammarAST targ = new GrammarAST(), c = new GrammarAST(), s = new
        GrammarAST();
    /* @atom11 */
    trackToken(targ);
    /* @atom5 */
    trackString(c);
    /* @atom6 */
    trackString(s);
  }
}
```

C.3 ANTLRWeaver_related.java

```
import org.antlr.tool.*;
import java.util.StringTokenizer;
import org.java.reusabledefinitions.JavaAnnotationNodesParser;

import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
```

```
import java.util.*;

public class ANTLRWeaver_related extends ANTLRWeaver_relatedBase {

  protected Grammar grammar;
  protected boolean showActions;
  protected StringBuilder buf = new StringBuilder(300);
  public String processingProduction = null;

  private ANTLRWeaver_generated.block_return block(GrammarAST t,
     boolean forceParens) throws RecognitionException {
    ANTLRWeaver_generated other = new ANTLRWeaver_generated(new
       CommonTreeNodeStream(t));
      other.buf = buf;
      return other.block(forceParens);
  }

  public final int countAltsForBlock(GrammarAST t) {
      int n = 0;
      for ( int i = 0; i < t.getChildCount(); i++ )
      {
          if ( t.getChild(i).getType() == ALT )
              n++;
      }

      return n;
  }

  public void out(String s) {
      buf.append(s);
  }

  @Override
  public void reportError(RecognitionException ex) {
      Token token = null;
      if (ex instanceof MismatchedTokenException) {
          token = ((MismatchedTokenException)ex).token;
      } else if (ex instanceof NoViableAltException) {
          token = ((NoViableAltException)ex).token;
      }

      ErrorManager.syntaxError(
          ErrorManager.MSG_SYNTAX_ERROR,
          grammar,
          token,
          "antlr.print:␣" + ex.toString(),
          ex );
  }

  /** Normalize a grammar print out by removing all double spaces
   *  and trailing/beginning stuff.  FOr example, convert
   *
```

```
 *  ( A  |  B  |  C )*
 *
 *  to
 *
 *  ( A | B | C )*
 */
public static String normalize(String g) {
    StringTokenizer st = new StringTokenizer(g, "␣", false);
    StringBuffer buf = new StringBuffer();
    while ( st.hasMoreTokens() ) {
        String w = st.nextToken();
        buf.append(w);
        buf.append("␣");
    }
    return buf.toString().trim();
}

 public static class modifier_return extends TreeRuleReturnScope {
    };

    public static class block_return extends TreeRuleReturnScope {
    };

    public static class atom_return extends TreeRuleReturnScope {
    };

public String prod_toString(Grammar g, Boolean showActions) throws
   RecognitionException{
  /* @returns */
  String s=null;
  /* @init */
  grammar = g;
  this.showActions = showActions;
  /* @toString1 */
  return normalize(buf.toString());
}

public void prod_grammar_() throws RecognitionException{
  /* @returns */
  String s=null;
  /* @grammar_1 */
  prod_grammarSpec("lexer");
  /* @grammar_2 */
  prod_grammarSpec("parser␣");
  /* @grammar_3 */
  prod_grammarSpec("tree␣");
  /* @grammar_4 */
  prod_grammarSpec("");
}

public void prod_grammarSpec (String gtype) throws
   RecognitionException{
  GrammarAST id = new GrammarAST(), cmt = new GrammarAST();
```

```
    /* @grammarSpec1 */
    out(gtype+"grammar␣
        "+JavaAnnotationNodesParser.className+"_generated;\n\n");
    /* @grammarSpec2 */
    out(cmt.getText()+"\n");
    /* @grammarSpec3 */
    out("\n");
  }

  public void prod_action () throws RecognitionException{
    GrammarAST id1 = new GrammarAST(), a1 = new GrammarAST(), a2 = new
        GrammarAST();
    /* @init */
    String scope=null, name=null;
    String action=null;
    /* @action1 */
    scope=id1.getText(); name=a1.getText(); action=a1.getText();
    /* @action2 */
    scope=null; name=id1.getText(); action=a2.getText();
    /* @action3 */
    out("\n@header␣{");
    out(action);
    for (String h : JavaAnnotationNodesParser.header){
            out(h+"\n");
      }
     out("}\n\n");

    out("\n@members␣{\n");
    for (String m : JavaAnnotationNodesParser.members){
            out(m+"\n\n");
      }
    out("}\n\n");
}

  public void prod_optionsSpec() throws RecognitionException{
    /* @optionsSpec1 */
    out("␣options␣{\n");
    /* @optionsSpec2 */
    out(";␣\n");
    /* @optionsSpec3 */
    out("}␣\n");
  }

  public void prod_option() throws RecognitionException{
    GrammarAST id = new GrammarAST();
    /* @option1 */
    out(id.getText()+"=");
  }

  public void prod_optionValue() throws RecognitionException{
    GrammarAST id = new GrammarAST(), s = new GrammarAST(), c = new
        GrammarAST(), i = new GrammarAST();
    /* @optionValue1 */
```

```
    out(id.getText());
    /* @optionValue2 */
    out(s.getText());
    /* @optionValue3 */
    out(c.getText());
    /* @optionValue4 */
    out(i.getText());
  }

  public void prod_rule() throws RecognitionException{
    GrammarAST id = new GrammarAST(), arg = new GrammarAST(), ret = new
       GrammarAST();
    /* @rule1 */
    out(id.getText());
    /* @rule10 */
    for (JavaAnnotationNodesParser.AnnotationContent c :
       JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.params != null)
              out(c.params);
            }
       }
    /* @rule11 */
    for (JavaAnnotationNodesParser.AnnotationContent c :
       JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.annTable.get("/* @returns */") != null)
              out(" " + c.annTable.get("/* @returns */"));
            }
       }
    /* @rule4 */
     for (JavaAnnotationNodesParser.AnnotationContent c :
         JavaAnnotationNodesParser.table){
              if (c.method.equals(processingProduction)){
                if (c.annTable.get("/* @init */") != null)
                  out("\n@init " + "{\n" + c.annTable.get("/* @init */")
                     + "}\n");

                if (c.annTable.get("/* @after */") != null)
                  out("\n@after " + "{\n" + c.annTable.get("/* @init
                     */") + "}\n");
           }
       }
    /* @rule5 */
    if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
      out(" ");
    /* @rule6 */
    out(";\n\n");
    /* @rule7 */
    prod_block(false);
    /* @rule8 */
    processingProduction = id.getText();
    /* @rule9 */
```

```
  for (JavaAnnotationNodesParser.AnnotationContent c :
     JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.annTable.get("/* @options */") != null)
              out("\noptions " + "{\n" + c.annTable.get("/* @options
                 */") + "}\n");
        }
    }
}

public void prod_precRule() throws RecognitionException{
  GrammarAST id = new GrammarAST(), arg = new GrammarAST(), ret = new
     GrammarAST();
  /* @precRule1 */
  out(id.getText());
  /* @precRule10 */
  for (JavaAnnotationNodesParser.AnnotationContent c :
     JavaAnnotationNodesParser.table){
        if (c.method.equals(processingProduction)){
          if (c.params != null)
            out(c.params);
          }
     }
  /* @precRule11 */
  for (JavaAnnotationNodesParser.AnnotationContent c :
     JavaAnnotationNodesParser.table){
        if (c.method.equals(processingProduction)){
          if (c.annTable.get("/* @returns */") != null)
            out(" " + c.annTable.get("/* @returns */"));
          }
     }
  /* @precRule4 */
   for (JavaAnnotationNodesParser.AnnotationContent c :
       JavaAnnotationNodesParser.table){
            if (c.method.equals(processingProduction)){
              if (c.annTable.get("/* @init */") != null)
                out("\n@init " + "{\n" + c.annTable.get("/* @init */")
                    + "}\n");

              if (c.annTable.get("/* @after */") != null)
                out("\n@after " + "{\n" + c.annTable.get("/* @init
                    */") + "}\n");
          }
      }
  /* @precRule5 */
  if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
    out(" ");
  /* @precRule6 */
  out(";\n\n");
  /* @precRule7 */
  prod_block(false);
  /* @precRule8 */
  processingProduction = id.getText();
```

```
  /* @precRule9 */
  for (JavaAnnotationNodesParser.AnnotationContent c :
     JavaAnnotationNodesParser.table){
          if (c.method.equals(processingProduction)){
            if (c.annTable.get("/*␣@options␣*/") != null)
              out("\noptions␣" + "{\n" + c.annTable.get("/*␣@options␣
                 */") + "}\n");
        }
    }

}


public void prod_modifier() throws RecognitionException{
  modifier_return retval = new modifier_return();
  /* @init */
  out(((GrammarAST) retval.start).getText()); out("␣");
}

public void prod_block(Boolean forceParens) throws
    RecognitionException{
  block_return retval = new block_return();
  /* @init */
  int numAlts = countAltsForBlock((GrammarAST) retval.start);
  /* @block1 */
  if ( forceParens||numAlts>1 )
  {
    out("␣(");
  }
  /* @block2 */
  out("␣:");
  /* @block3 */
  out("\n|");
  /* @block4 */
  if ( forceParens||numAlts>1 ) out(")");
}

public void prod_singleRewrite() throws RecognitionException{
  GrammarAST s = new GrammarAST(), a = new GrammarAST();
  /* @singleRewrite1 */
  out("␣->");
  /* @singleRewrite2 */
  out("␣{"+s.getText()+"}?");
  /* @singleRewrite3 */
  out("...");
  /* @singleRewrite4 */
  out("␣{"+a.getText()+"}");
}

public void prod_rewriteTemplate() throws RecognitionException{
  GrammarAST id = new GrammarAST(), ind = new GrammarAST(), arg = new
     GrammarAST(),
```

```
        a = new GrammarAST(), dq = new GrammarAST(), da = new
            GrammarAST() ;
    /* @rewriteTemplate1 */
    out("␣"+id.getText());
    /* @rewriteTemplate2 */
    out("␣({"+ind.getText()+"})");
    /* @rewriteTemplate3 */
    out("(");
    /* @rewriteTemplate4 */
    out(arg.getText()+"=");
    /* @rewriteTemplate5 */
    out(a.getText());
    /* @rewriteTemplate6 */
    out(")");
    /* @rewriteTemplate7 */
    out("␣"+dq.getText());
    /* @rewriteTemplate8 */
    out("␣"+da.getText());
  }

  public void prod_element() throws RecognitionException{
    GrammarAST id = new GrammarAST(), id2 = new GrammarAST(), a = new
        GrammarAST(),
        a2 = new GrammarAST(), pred = new GrammarAST(), spred = new
            GrammarAST(),
        gpred = new GrammarAST(), ann = new GrammarAST();
    /* @element1 */
    out("~");
    /* @element2 */
    out("!");
    /* @element3 */
    out("~");
    /* @element4 */
    out("..");
    /* @element5 */
    out("..");
    /* @element6 */
    out(id.getText()+"=");
    /* @element7 */
    out(id2.getText()+"+=");
    /* @element8 */
    out("=>");
    /* @element9 */
    out("{"); out(a.getText()); out("}");
    /* @element10 */
    out("{{"); out(a2.getText()); out("}}");
    /* @element11 */
    if ( showActions )
    {
      out("{");
      out(pred.getText());
      out("}?");
    }
```

```
    else
    {
      out("{...}?");
    }
    /* @element12 */
    String name = spred.getText();
    GrammarAST predAST=grammar.getSyntacticPredicate(name);
    block(predAST, true);
    out("=>");
    /* @element13 */
    if ( showActions ) {out("{"); out(gpred.getText()); out("}?␣=>");}
    else {out("{...}?␣=>");}
    /* @element14 */
    prod_block(true);
    /* @element15 */
     for (JavaAnnotationNodesParser.AnnotationContent c :
         JavaAnnotationNodesParser.table){
              if (c.method.equals(processingProduction)){
                if (c.annTable.get(ann.toString()) != null) {
                  if (c.annTable.get(ann.toString()).charAt(0) == '[')
                  out(c.annTable.get(ann.toString()));
                  else
                  out("{" + c.annTable.get(ann.toString()) + "}\n");
                }
            }
       }

  }

  public void prod_ebnf() throws RecognitionException{
    /* @ebnf1 */
    out("␣");
    /* @ebnf2 */
    out("?␣");
    /* @ebnf3 */
    out("*␣");
    /* @ebnf4 */
    out("+␣");
    /* @ebnf5 */
    prod_block(true);
    /* @ebnf6 */
    prod_block(true);
    /* @ebnf7 */
    prod_block(true);
    /* @ebnf8 */
    prod_block(true);

  }

  public void prod_tree() throws RecognitionException{
    /* @tree1 */
    out("␣^(");
    /* @tree2 */
```

```
    out(")␣");
  }

  public void prod_atom() throws RecognitionException{
    atom_return retval = new atom_return();
    GrammarAST rarg = new GrammarAST(), targ = new GrammarAST(), l =
       new GrammarAST(),
        id = new GrammarAST();
    /* @init */
    out("␣");
    /* @atom1 */
    out(((GrammarAST) retval.start).toString());
    /* @atom2 */
    out("["+rarg.toString()+"]");
    /* @atom3 */
    out(((GrammarAST) retval.start).toString());
    /* @atom4 */
    out("["+targ.toString()+"]");
    /* @atom5 */
    out(((GrammarAST) retval.start).toString());
    /* @atom6 */
    out(((GrammarAST) retval.start).toString());
    /* @atom7 */
    out(((GrammarAST) retval.start).toString());
    /* @atom8 */
    out("␣");
    /* @atom9 */
    out("␣$"+l.getText());
    /* @atom10 */
    out(id.getText()+".");
  }

  public void prod_ast_suffix() throws RecognitionException{
    /* @ast_suffix1 */
    out("^");
    /* @ast_suffix2 */
    out("!");
  }
}
```

Appendix D

Generated Grammars

This Appendix contains all grammars generated by our weaver after using the related classes from Appendix C and the general grammar from Appendix B.

D.1 ANTLRTreePrinter_generated.g

```
tree grammar ANTLRTreePrinter_generated;

 options {
tokenVocab=ANTLR;
ASTLabelType=GrammarAST;
}


@header {
package org.antlr.grammar.v3;
import org.antlr.tool.*;
import java.util.StringTokenizer;
}


@members {
protected Grammar grammar;

protected boolean showActions;

protected StringBuilder buf = new StringBuilder(300);

private ANTLRTreePrinter.block_return block (GrammarAST t, boolean
   forceParens) throws RecognitionException{
   ANTLRTreePrinter other = new ANTLRTreePrinter(new
      CommonTreeNodeStream(t));
```

```
   other.buf = buf;
   return other.block(forceParens);
  }

public final int countAltsForBlock (GrammarAST t){
   int n = 0;
   for ( int i = 0; i < t.getChildCount(); i++ )
   {
   if ( t.getChild(i).getType() == ALT )
   n++;
   }

   return n;
  }

public void out (String s){
   buf.append(s);
  }

@Override
  public void reportError (RecognitionException ex){
   Token token = null;
   if (ex instanceof MismatchedTokenException) {
   token = ((MismatchedTokenException)ex).token;
   } else if (ex instanceof NoViableAltException) {
   token = ((NoViableAltException)ex).token;
   }

   ErrorManager.syntaxError(
   ErrorManager.MSG_SYNTAX_ERROR,
   grammar,
   token,
   "antlr.print:␣" + ex.toString(),
   ex );
  }

public static String normalize (String g){

   StringTokenizer st = new StringTokenizer(g, "␣", false);
   StringBuffer buf = new StringBuffer();
   while ( st.hasMoreTokens() ) {
   String w = st.nextToken();
   buf.append(w);
   buf.append("␣");
   }
   return buf.toString().trim();
  }

}

public
toString[Grammar g, boolean showActions] returns [String s=null]
@init {
```

```
  grammar = g;
  this.showActions = showActions;
}
  : ( grammar_
    | rule
    | alternative
    | element
    | single_rewrite
    | rewrite
    | EOR //{s="EOR";}
    )
    {return normalize(buf.toString());}
  ;

public grammar_ returns [String s=null]
 :
 ( ^( LEXER_GRAMMAR grammarSpec ["lexer"]
)
| ^( PARSER_GRAMMAR grammarSpec ["parser␣"]
)
| ^( TREE_GRAMMAR grammarSpec ["tree␣"]
)
| ^( COMBINED_GRAMMAR grammarSpec [""]
) ) ;

grammarSpec[String gtype] :
 id= ID {out(gtype+"grammar␣"+id.getText());
}
 (cmt= DOC_COMMENT {out(cmt.getText()+"\n");
}
)? ( optionsSpec )? {out(";\n");
}
 ( delegateGrammars )? ( tokensSpec )? ( attrScope )* ( actions )?
    rules ;

attrScope :
 ^( 'scope' ID ( attrScopeAction )* ACTION ) ;

attrScopeAction :
 ^( AMPERSAND ID ACTION ) ;

actions :
 ( action )+ ;

action
@init {
String scope=null, name=null;
String action=null;
}
 :
 ^( AMPERSAND id1= ID (id2= ID a1= ACTION {scope=id1.getText();
name=a1.getText();
action=a1.getText();
```

```
}

|a2= ACTION {scope=null;
name=id1.getText();
action=a2.getText();
}
) ) {if ( showActions )
    {
      out("@"+(scope!=null?scope+"::":"")+name+action);
    }
}
;

optionsSpec :
 ^( OPTIONS {out("␣options␣{");
}
 ( option {out(";␣");
}
)+ {out("}␣");
}
) ;

option :
 ^( ASSIGN id= ID {out(id.getText()+"=");
}
o= optionValue ) ;

optionValue :
 (id= ID {out(id.getText());
}

|s= STRING_LITERAL {out(s.getText());
}

|c= CHAR_LITERAL {out(c.getText());
}

|i= INT {out(i.getText());
}
);

delegateGrammars :
 ^( 'import' ( ^( ASSIGN ID ID )
| ID )+ ) ;

tokensSpec :
 ^( TOKENS ( tokenSpec )* ) ;

tokenSpec :
 (t= TOKEN_REF
| ^( ASSIGN t2= TOKEN_REF ( STRING_LITERAL
| CHAR_LITERAL ) ) );
```

```
rules :
 ( rule
| precRule )+ ;

rule :
 ^( RULE id= ID ( modifier )? {out(id.getText());
}
 ^( ARG (arg= ARG_ACTION {out("["+arg.getText()+"]");
}
)? ) ^( RET (ret= ARG_ACTION {out("␣returns␣["+ret.getText()+"]");
}
)? ) ( throwsSpec )? ( optionsSpec )? ( ruleScopeSpec )? ( ruleAction
   )* {out("␣:");
}
{if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
      out("␣");
}
b= block [false]
 ( exceptionGroup )? EOR {out(";\n");
}
) ;

precRule :
 ^( PREC_RULE id= ID ( modifier )? {out(id.getText());
}
 ^( ARG (arg= ARG_ACTION {out("["+arg.getText()+"]");
}
)? ) ^( RET (ret= ARG_ACTION {out("␣returns␣["+ret.getText()+"]");
}
)? ) ( throwsSpec )? ( optionsSpec )? ( ruleScopeSpec )? ( ruleAction
   )* {out("␣:");
}
{if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
      out("␣");
}
b= block [false]
 ( exceptionGroup )? EOR {out(";\n");
}
) ;

ruleAction :
 ^( AMPERSAND id= ID a= ACTION ) {if ( showActions )
    out("@"+id.getText()+"{"+a.getText()+"}");
}
;

modifier
@init {
out(($start).getText());
out("␣");
}
 :
 ( 'protected'
```

```
| 'public'
| 'private'
| 'fragment' );

throwsSpec :
 ^( 'throws' ( ID )+ ) ;

ruleScopeSpec :
 ^( 'scope' ( attrScopeAction )* ( ACTION )? ( ID )* ) ;

block[Boolean forceParens]
@init {
int numAlts = countAltsForBlock($start);
}
 :
 ^( BLOCK {if ( forceParens||numAlts>1 )
    {
      out("␣(");
    }
}
 ( optionsSpec {out("␣:");
}
)? ( alternative rewrite {out("|");
}
)+ EOB {if ( forceParens||numAlts>1 ) out(")");
}
) ;

alternative :
 ^( ALT ( element )* EOA ) ;

exceptionGroup :
 ( ( exceptionHandler )+ ( finallyClause )?
| finallyClause );

exceptionHandler :
 ^( 'catch' ARG_ACTION ACTION ) ;

finallyClause :
 ^( 'finally' ACTION ) ;

rewrite :
 ( ^( REWRITES ( single_rewrite )+ )
| REWRITES
|);

single_rewrite :
 ^( REWRITE ( SEMPRED )? ( rewrite_alternative
| rewrite_template
| ETC
| ACTION ) ) ;

rewrite_alternative :
```

```
 ^(a= ALT ( ( rewrite_element )+
| EPSILON ) EOA ) ;

rewrite_element :
 ( rewrite_atom
| rewrite_ebnf
| rewrite_tree );

rewrite_ebnf :
 ( ^( OPTIONAL rewrite_block )
| ^( CLOSURE rewrite_block )
| ^( POSITIVE_CLOSURE rewrite_block ) );

rewrite_block :
 ^( BLOCK ( rewrite_alternative
| rewrite_template
| ETC ) EOB ) ;

rewrite_tree :
 ^( TREE_BEGIN rewrite_atom ( rewrite_element )* ) ;

rewrite_atom :
 ( RULE_REF
| ( ^( TOKEN_REF ( ARG_ACTION )? )
| CHAR_LITERAL
| STRING_LITERAL )
| LABEL
| ACTION );

rewrite_template :
 ^( TEMPLATE (id= ID
|ind= ACTION ) ^( ARGLIST ( ^( ARG arg= ID a= ACTION ) )* ) (dq=
   DOUBLE_QUOTE_STRING_LITERAL
|da= DOUBLE_ANGLE_STRING_LITERAL )? ) ;

element :
 ( ^( ROOT element ) {out("^");
}

| ^( BANG element ) {out("!");
}

| atom
| ^( NOT {out("~");
}
 element )
| ^( RANGE atom {out("..");
}
 atom )
| ^( CHAR_RANGE atom {out("..");
}
 atom )
| ^( ASSIGN id= ID {out(id.getText()+"=");
```

```
}
 element )
| ^( PLUS_ASSIGN id2= ID {out(id2.getText()+"+=");
}
 element )
| ebnf
| tree_
| ^( SYNPRED block [true]
) {out("=>");
}

|a= ACTION {out("{");
out(a.getText());
out("}");
}

|a2= FORCED_ACTION {out("{{");
out(a2.getText());
out("}}");
}

|pred= SEMPRED {if ( showActions )
    {
      out("{");
      out(pred.getText());
      out("}?");
    }
    else
    {
      out("{...}?");
    }
}

|spred= SYN_SEMPRED {String name = spred.getText();
GrammarAST predAST=grammar.getSyntacticPredicate(name);
block(predAST, true);
out("=>");
}

| ^( BACKTRACK_SEMPRED ( . )* )
|gpred= GATED_SEMPRED {if ( showActions ) {out("{");
   out(gpred.getText()); out("}?␣=>");}
    else {out("{...}?␣=>");}
}

| EPSILON );

ebnf :
 ( ( dotLoop )=> dotLoop
| block [true]
{out("␣");
}
```

```
| ^( OPTIONAL block [true]
) {out("?␣");
}

| ^( CLOSURE block [true]
) {out("*␣");
}

| ^( POSITIVE_CLOSURE block [true]
) {out("+␣");
}
);

dotLoop :
 ( ^( CLOSURE dotBlock )
| ^( POSITIVE_CLOSURE dotBlock ) ) ;

dotBlock :
 ^( BLOCK ^( ALT WILDCARD EOA ) EOB ) ;

tree_ :
 ^( TREE_BEGIN element ( element )* ) ;

atom
@init {
out("␣");
}
 :
 ( ( ^( RULE_REF {out(($start).toString());
}
 (rarg= ARG_ACTION {out("["+rarg.toString()+"]");
}
)? ( ast_suffix )? )
| ^( TOKEN_REF {out(($start).toString());
}
 (targ= ARG_ACTION {out("["+targ.toString()+"]");
}
)? ( ast_suffix )? ) {out("␣");
}

| ^( CHAR_LITERAL {out(($start).toString());
}
 ( ast_suffix )? )
| ^( STRING_LITERAL {out(($start).toString());
}
 ( ast_suffix )? )
| ^( WILDCARD {out(($start).toString());
}
 ( ast_suffix )? ) ) {out("␣");
}

|l= LABEL {out("␣$"+l.getText());
}
```

```
| ^( DOT id= ID {out(id.getText()+".");
}
 atom ) );

ast_suffix :
 ( ROOT {out("^");
}

| BANG {out("^");
}
);
```

D.2 AssignTokenTypesWalker_generated.g

```
tree grammar AssignTokenTypesWalker_generated;

 options {
tokenVocab=ANTLR;
ASTLabelType=GrammarAST;
}


@header {
package org.antlr.grammar.v3;
import org.antlr.tool.*;
import org.antlr.analysis.*;
import org.antlr.misc.*;
import org.antlr.runtime.BitSet;
import java.util.Stack;
import java.util.List;
import java.util.ArrayList;
}


@members {
protected Grammar grammar;

protected String currentRuleName;

protected static GrammarAST stringAlias;

protected static GrammarAST charAlias;

protected static GrammarAST stringAlias2;

protected static GrammarAST charAlias2;

@Override
  public void reportError (RecognitionException ex){
```

```
   Token token = null;
   if (ex instanceof MismatchedTokenException) {
   token = ((MismatchedTokenException)ex).token;
   } else if (ex instanceof NoViableAltException) {
   token = ((NoViableAltException)ex).token;
   }

   ErrorManager.syntaxError(
   ErrorManager.MSG_SYNTAX_ERROR,
   grammar,
   token,
   "assign.types:␣" + ex.toString(),
   ex);
  }

protected void initASTPatterns (){
   TreeAdaptor adaptor = new ANTLRParser.grammar_Adaptor(null);

   /*
   * stringAlias = ^(BLOCK[] ^(ALT[] STRING_LITERAL[] EOA[]) EOB[])
   */
   stringAlias = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
   {
   GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
   adaptor.addChild( alt, adaptor.create( STRING_LITERAL,
      "STRING_LITERAL" ) );
   adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
   adaptor.addChild( stringAlias, alt );
   }
   adaptor.addChild( stringAlias, adaptor.create( EOB, "EOB" ) );

   /*
   * charAlias = ^(BLOCK[] ^(ALT[] CHAR_LITERAL[] EOA[]) EOB[])
   */
   charAlias = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
   {
   GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
   adaptor.addChild( alt, adaptor.create( CHAR_LITERAL, "CHAR_LITERAL"
      ) );
   adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
   adaptor.addChild( charAlias, alt );
   }
   adaptor.addChild( charAlias, adaptor.create( EOB, "EOB" ) );

   /*
   * stringAlias2 = ^(BLOCK[] ^(ALT[] STRING_LITERAL[] ACTION[] EOA[])
      EOB[])
   */
   stringAlias2 = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
   {
   GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
   adaptor.addChild( alt, adaptor.create( STRING_LITERAL,
      "STRING_LITERAL" ) );
```

```
   adaptor.addChild( alt, adaptor.create( ACTION, "ACTION" ) );
   adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
   adaptor.addChild( stringAlias2, alt );
   }
   adaptor.addChild( stringAlias2, adaptor.create( EOB, "EOB" ) );

   /*
   * charAlias = ^(BLOCK[] ^(ALT[] CHAR_LITERAL[] ACTION[] EOA[]) EOB[])
   */
   charAlias2 = (GrammarAST)adaptor.create( BLOCK, "BLOCK" );
   {
   GrammarAST alt = (GrammarAST)adaptor.create( ALT, "ALT" );
   adaptor.addChild( alt, adaptor.create( CHAR_LITERAL, "CHAR_LITERAL"
      ) );
   adaptor.addChild( alt, adaptor.create( ACTION, "ACTION" ) );
   adaptor.addChild( alt, adaptor.create( EOA, "EOA" ) );
   adaptor.addChild( charAlias2, alt );
   }
   adaptor.addChild( charAlias2, adaptor.create( EOB, "EOB" ) );
  }

protected void trackString (GrammarAST t){}

protected void trackToken ( GrammarAST t ){}

protected void trackTokenRule ( GrammarAST t, GrammarAST modifier,
   GrammarAST block ){}

protected void alias ( GrammarAST t, GrammarAST s ){}

public void defineTokens ( Grammar root ){}

protected void defineStringLiteralsFromDelegates (){}

protected void assignStringTypes ( Grammar root ){}

protected void aliasTokenIDsAndLiterals ( Grammar root ){}

protected void assignTokenIDTypes ( Grammar root ){}

protected void defineTokenNamesAndLiteralsInGrammar ( Grammar root ){}

protected void init ( Grammar root ){}

public Object value=null;

}

public grammar_[Grammar g]
@init {
if ( state.backtracking == 0 )
      init(g);
}
```

```
 :
 ( ^( LEXER_GRAMMAR grammarSpec )
| ^( PARSER_GRAMMAR grammarSpec )
| ^( TREE_GRAMMAR grammarSpec )
| ^( COMBINED_GRAMMAR grammarSpec ) ) ;

grammarSpec :
 id= ID (cmt= DOC_COMMENT )? ( optionsSpec )? ( delegateGrammars )? (
    tokensSpec )? ( attrScope )* ( actions )? rules ;

attrScope :
 ^( 'scope' ID ( attrScopeAction )* ACTION ) ;

attrScopeAction :
 ^( AMPERSAND ID ACTION ) ;

actions :
 ( action )+ ;

action :
 ^( AMPERSAND id1= ID (id2= ID a1= ACTION
|a2= ACTION ) ) ;

optionsSpec returns [Map<Object, Object> opts = new HashMap<Object,
   Object>()]
 :
 ^( OPTIONS ( option [opts]
)+ ) ;

option[Map<Object, Object> opts] :
 ^( ASSIGN id= ID o= optionValue ) {String key = id.getText();
opts.put(key, o.value);
if ( currentRuleName==null && key.equals("tokenVocab") )
    {
      grammar.importTokenVocabulary(id,(String)o.value);
    }
}
;

optionValue returns [Object value=null]

@init {
if ( state.backtracking == 0 )
      retval.value = ($start).getText();
}
 :
 (id= ID
|s= STRING_LITERAL
|c= CHAR_LITERAL
|i= INT {retval.value = Integer.parseInt(i.getText());
}
);
```

```
delegateGrammars :
 ^( 'import' ( ^( ASSIGN ID ID )
| ID )+ ) ;

tokensSpec :
 ^( TOKENS ( tokenSpec )* ) ;

tokenSpec :
 (t= TOKEN_REF {trackToken(t);
}

| ^( ASSIGN t2= TOKEN_REF {trackToken(t2);
}
 (s= STRING_LITERAL {trackString(s);
alias(t2,s);
}

|c= CHAR_LITERAL {trackString(c);
alias(t2,c);
}
) ) );

rules :
 ( rule
| precRule )+ ;

rule :
 ^( RULE id= ID {currentRuleName=id.getText();
}
 (m= modifier )? ^( ARG (arg= ARG_ACTION )? ) ^( RET (ret= ARG_ACTION
    )? ) ( throwsSpec )? ( optionsSpec )? ( ruleScopeSpec )? (
    ruleAction )* b= block ( exceptionGroup )? EOR
    {trackTokenRule(id,$m.start,$b.start);
}
) ;

precRule :
 ^( PREC_RULE id= ID {currentRuleName=id.getText();
}
 (m= modifier )? ^( ARG (arg= ARG_ACTION )? ) ^( RET (ret= ARG_ACTION
    )? ) ( throwsSpec )? ( optionsSpec )? ( ruleScopeSpec )? (
    ruleAction )* b= block ( exceptionGroup )? EOR
    {trackTokenRule(id,$m.start,$b.start);
}
) ;

ruleAction :
 ^( AMPERSAND id= ID a= ACTION ) ;

modifier :
 ( 'protected'
| 'public'
| 'private'
```

```
| 'fragment' );

throwsSpec :
 ^( 'throws' ( ID )+ ) ;

ruleScopeSpec :
 ^( 'scope' ( attrScopeAction )* ( ACTION )? ( ID )* ) ;

block :
 ^( BLOCK ( optionsSpec )? ( alternative rewrite )+ EOB ) ;

alternative :
 ^( ALT ( element )* EOA ) ;

exceptionGroup :
 ( ( exceptionHandler )+ ( finallyClause )?
| finallyClause );

exceptionHandler :
 ^( 'catch' ARG_ACTION ACTION ) ;

finallyClause :
 ^( 'finally' ACTION ) ;

rewrite :
 ( ^( REWRITES ( single_rewrite )+ )
| REWRITES
|);

single_rewrite :
 ^( REWRITE ( SEMPRED )? ( rewrite_alternative
| rewrite_template
| ETC
| ACTION ) ) ;

rewrite_alternative :
 ^(a= ALT ( ( rewrite_element )+
| EPSILON ) EOA ) ;

rewrite_element :
 ( rewrite_atom
| rewrite_ebnf
| rewrite_tree );

rewrite_ebnf :
 ( ^( OPTIONAL rewrite_block )
| ^( CLOSURE rewrite_block )
| ^( POSITIVE_CLOSURE rewrite_block ) );

rewrite_block :
 ^( BLOCK ( rewrite_alternative
| rewrite_template
| ETC ) EOB ) ;
```

```
rewrite_tree :
 ^( TREE_BEGIN rewrite_atom ( rewrite_element )* ) ;

rewrite_atom :
 ( RULE_REF
| ( ^( TOKEN_REF ( ARG_ACTION )? )
| CHAR_LITERAL
| STRING_LITERAL )
| LABEL
| ACTION );

rewrite_template :
 ^( TEMPLATE (id= ID
|ind= ACTION ) ^( ARGLIST ( ^( ARG arg= ID a= ACTION ) )* ) (dq=
   DOUBLE_QUOTE_STRING_LITERAL
|da= DOUBLE_ANGLE_STRING_LITERAL )? ) ;

element :
 ( ^( ROOT element )
| ^( BANG element )
| atom
| ^( NOT element )
| ^( RANGE atom atom )
| ^( CHAR_RANGE atom atom )
| ^( ASSIGN id= ID element )
| ^( PLUS_ASSIGN id2= ID element )
| ebnf
| tree_
| ^( SYNPRED block )
|a= ACTION
|a2= FORCED_ACTION
|pred= SEMPRED
|spred= SYN_SEMPRED
| ^( BACKTRACK_SEMPRED ( . )* )
|gpred= GATED_SEMPRED
| EPSILON );

ebnf :
 ( ( dotLoop )=> dotLoop
| block
| ^( OPTIONAL block )
| ^( CLOSURE block )
| ^( POSITIVE_CLOSURE block ) );

dotLoop :
 ( ^( CLOSURE dotBlock )
| ^( POSITIVE_CLOSURE dotBlock ) ) ;

dotBlock :
 ^( BLOCK ^( ALT WILDCARD EOA ) EOB ) ;

tree_ :
```

```
 ^( TREE_BEGIN element ( element )* ) ;

atom :
 ( ( ^( RULE_REF (rarg= ARG_ACTION )? ( ast_suffix )? )
| ^( TOKEN_REF (targ= ARG_ACTION )? ( ast_suffix )? ) {trackToken(targ);
}

| ^(c= CHAR_LITERAL {trackString(c);
}
 ( ast_suffix )? )
| ^(s= STRING_LITERAL {trackString(s);
}
 ( ast_suffix )? )
| ^( WILDCARD ( ast_suffix )? ) )
|l= LABEL
| ^( DOT id= ID atom ) );

ast_suffix :
 ( ROOT
| BANG );
```

D.3 ANTLRWeaver_generated.g

```
tree grammar ANTLRWeaver_generated;

 options {
tokenVocab=ANTLRAnnotationNodes;
ASTLabelType=GrammarAST;
}


@header {
package org.antlr.grammar.v3;
import org.antlr.tool.*;
import java.util.StringTokenizer;
import org.java.reusabledefinitions.JavaAnnotationNodesParser;
}


@members {
protected Grammar grammar;

protected boolean showActions;

protected StringBuilder buf = new StringBuilder(300);

public String processingProduction = null;

private ANTLRWeaver_generated.block_return block (GrammarAST t, boolean
   forceParens) throws RecognitionException{
```

```
    ANTLRWeaver_generated other = new ANTLRWeaver_generated(new
       CommonTreeNodeStream(t));
   other.buf = buf;
   return other.block(forceParens);
  }

public final int countAltsForBlock (GrammarAST t){
   int n = 0;
   for ( int i = 0; i < t.getChildCount(); i++ )
   {
   if ( t.getChild(i).getType() == ALT )
   n++;
   }

   return n;
  }

public void out (String s){
   buf.append(s);
  }

@Override
  public void reportError (RecognitionException ex){
   Token token = null;
   if (ex instanceof MismatchedTokenException) {
   token = ((MismatchedTokenException)ex).token;
   } else if (ex instanceof NoViableAltException) {
   token = ((NoViableAltException)ex).token;
   }

   ErrorManager.syntaxError(
   ErrorManager.MSG_SYNTAX_ERROR,
   grammar,
   token,
   "antlr.print:␣" + ex.toString(),
   ex );
  }

public static String normalize (String g){
   StringTokenizer st = new StringTokenizer(g, "␣", false);
   StringBuffer buf = new StringBuffer();
   while ( st.hasMoreTokens() ) {
   String w = st.nextToken();
   buf.append(w);
   buf.append("␣");
   }
   return buf.toString().trim();
  }

}

/** Call this to figure out how to print */
public
```

```
toString[Grammar g, boolean showActions] returns [String s=null]
@init {
  grammar = g;
  this.showActions = showActions;
}
  : ( grammar_
    | rule
    | alternative
    | element
    | single_rewrite
    | rewrite
    | EOR //{s="EOR";}
    )
    {return normalize(buf.toString());}
  ;

// --------------

public grammar_ returns [String s=null]
 :
 ( ^( LEXER_GRAMMAR grammarSpec ["lexer"]
)
| ^( PARSER_GRAMMAR grammarSpec ["parser␣"]
)
| ^( TREE_GRAMMAR grammarSpec ["tree␣"]
)
| ^( COMBINED_GRAMMAR grammarSpec [""]
) ) ;

grammarSpec[String gtype] :
 id= ID {out(gtype+"grammar␣
    "+JavaAnnotationNodesParser.className+"_generated;\n\n");
}
 (cmt= DOC_COMMENT {out(cmt.getText()+"\n");
}
)? ( optionsSpec )? {out("\n");
}
 ( delegateGrammars )? ( tokensSpec )? ( attrScope )* ( actions )?
    rules ;

attrScope :
 ^( 'scope' name= ID ( attrScopeAction )* attrs= ACTION ) ;

attrScopeAction :
 ^( AMPERSAND id= ID a= ACTION ) ;

actions :
 ( action )+ ;

action
@init {
String scope=null, name=null;
String action=null;
```

```
}
 :
 ^(amp= AMPERSAND id1= ID (id2= ID a1= ACTION {scope=id1.getText();
name=a1.getText();
action=a1.getText();
}

|a2= ACTION {scope=null;
name=id1.getText();
action=a2.getText();
}
) ) {out("\n@header␣{");
out(action);
for (String h : JavaAnnotationNodesParser.header){
     out(h+"\n");
     }
out("}\n\n");
out("\n@members␣{\n");
for (String m : JavaAnnotationNodesParser.members){
     out(m+"\n\n");
     }
out("}\n\n");
}
;

optionsSpec :
 ^( OPTIONS {out("␣options␣{\n");
}
 ( option {out(";␣\n");
}
)+ {out("}␣\n");
}
) ;

option :
 ^( ASSIGN id= ID {out(id.getText()+"=");
}
o= optionValue ) ;

optionValue :
 (id= ID {out(id.getText());
}

|s= STRING_LITERAL {out(s.getText());
}

|c= CHAR_LITERAL {out(c.getText());
}

|i= INT {out(i.getText());
}
);
```

```
delegateGrammars :
 ^( 'import' ( ^( ASSIGN ID ID )
| ID )+ ) ;

tokensSpec :
 ^( TOKENS ( tokenSpec )* ) ;

tokenSpec :
 (t= TOKEN_REF
| ^( ASSIGN t2= TOKEN_REF (s= STRING_LITERAL
|c= CHAR_LITERAL ) ) );

rules :
 ( rule
| precRule )+ ;

rule :
 ^(ru= RULE id= ID {processingProduction = id.getText();
}
 (m= modifier )? {out(id.getText());
}
 ^( ARG {for (JavaAnnotationNodesParser.AnnotationContent c :
    JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.params != null)
   out(c.params);
   }
   }
}
 (arg= ARG_ACTION )? ) ^( RET {for
    (JavaAnnotationNodesParser.AnnotationContent c :
    JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.annTable.get("/* @returns */") != null)
   out(" " + c.annTable.get("/* @returns */"));
   }
   }
}
 (ret= ARG_ACTION )? ) (t= throwsSpec )? ( optionsSpec )? {for
    (JavaAnnotationNodesParser.AnnotationContent c :
    JavaAnnotationNodesParser.table){
 if (c.method.equals(processingProduction)){
 if (c.annTable.get("/* @options */") != null)
 out("\noptions " + "{\n" + c.annTable.get("/* @options */") + "}\n");
 }
 }
}
 ( ruleScopeSpec {for (JavaAnnotationNodesParser.AnnotationContent c :
    JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.params != null)
   out(c.params);
   }
```

```
   }
}
)? ( ruleAction {for (JavaAnnotationNodesParser.AnnotationContent c :
   JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.annTable.get("/*␣@returns␣*/") != null)
   out("␣" + c.annTable.get("/*␣@returns␣*/"));
   }
   }
}
)* {for (JavaAnnotationNodesParser.AnnotationContent c :
   JavaAnnotationNodesParser.table){
 if (c.method.equals(processingProduction)){
 if (c.annTable.get("/*␣@init␣*/") != null)
 out("\n@init␣" + "{\n" + c.annTable.get("/*␣@init␣*/") + "}\n");

 if (c.annTable.get("/*␣@after␣*/") != null)
 out("\n@after␣" + "{\n" + c.annTable.get("/*␣@init␣*/") + "}\n");
 }
 }
}
{if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
      out("␣");
}
b= block [false]
 ( exceptionGroup )? EOR {out(";\n\n");
}
) ;

precRule :
 ^( PREC_RULE id= ID {processingProduction = id.getText();
}
 (m= modifier )? {out(id.getText());
}
 ^( ARG {for (JavaAnnotationNodesParser.AnnotationContent c :
    JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.params != null)
   out(c.params);
   }
   }
}
 (arg= ARG_ACTION )? ) ^( RET {for
    (JavaAnnotationNodesParser.AnnotationContent c :
    JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.annTable.get("/*␣@returns␣*/") != null)
   out("␣" + c.annTable.get("/*␣@returns␣*/"));
   }
   }
}
 (ret= ARG_ACTION )? ) ( throwsSpec )? ( optionsSpec )? {for
    (JavaAnnotationNodesParser.AnnotationContent c :
```

```
    JavaAnnotationNodesParser.table){
 if (c.method.equals(processingProduction)){
 if (c.annTable.get("/* @options */") != null)
 out("\noptions " + "{\n" + c.annTable.get("/* @options */") + "}\n");
 }
 }
}
 ( ruleScopeSpec {for (JavaAnnotationNodesParser.AnnotationContent c :
    JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.params != null)
   out(c.params);
   }
   }
}
)? ( ruleAction {for (JavaAnnotationNodesParser.AnnotationContent c :
   JavaAnnotationNodesParser.table){
   if (c.method.equals(processingProduction)){
   if (c.annTable.get("/* @returns */") != null)
   out(" " + c.annTable.get("/* @returns */"));
   }
   }
}
)* {for (JavaAnnotationNodesParser.AnnotationContent c :
   JavaAnnotationNodesParser.table){
 if (c.method.equals(processingProduction)){
 if (c.annTable.get("/* @init */") != null)
 out("\n@init " + "{\n" + c.annTable.get("/* @init */") + "}\n");

 if (c.annTable.get("/* @after */") != null)
 out("\n@after " + "{\n" + c.annTable.get("/* @init */") + "}\n");
 }
 }
}
{if ( input.LA(5) == NOT || input.LA(5) == ASSIGN )
      out(" ");
}
b= block [false]
 ( exceptionGroup )? EOR {out(";\n\n");
}
) ;

ruleAction :
 ^(amp= AMPERSAND id= ID a= ACTION ) ;

modifier
@init {
out(($start).getText());
out(" ");
}
 :
 ( 'protected'
| 'public'
```

```
| 'private'
| 'fragment' );

throwsSpec :
 ^( 'throws' (id= ID )+ ) ;

ruleScopeSpec :
 ^( 'scope' ( attrScopeAction )* (attrs= ACTION )? (uses= ID )* ) ;

block[Boolean forceParens]
@init {
int numAlts = countAltsForBlock($start);
}
 :
 ^( BLOCK {if ( forceParens||numAlts>1 )
    {
      out("␣(");
    }
}
 ( optionsSpec {out("␣:");
}
)? ( alternative rewrite {out("\n|");
}
)+ EOB {if ( forceParens||numAlts>1 ) out(")");
}
) ;

alternative :
 ^( ALT ( element )* EOA ) ;

exceptionGroup :
 ( ( exceptionHandler )+ ( finallyClause )?
| finallyClause );

exceptionHandler :
 ^( 'catch' ARG_ACTION a= ACTION ) ;

finallyClause :
 ^( 'finally' a= ACTION ) ;

rewrite :
 ( ^( REWRITES ( single_rewrite )+ )
| REWRITES
|);

single_rewrite :
 ^( REWRITE (pred= SEMPRED )? ( rewrite_alternative
| rewrite_template
| ETC
|act= ACTION ) ) ;

rewrite_alternative :
 ^(a= ALT ( ( rewrite_element )+
```

```
| EPSILON ) EOA ) ;

rewrite_element :
 ( rewrite_atom
| rewrite_ebnf
| rewrite_tree );

rewrite_ebnf :
 ( ^( OPTIONAL rewrite_block )
| ^( CLOSURE rewrite_block )
| ^( POSITIVE_CLOSURE rewrite_block ) );

rewrite_block :
 ^( BLOCK ( rewrite_alternative
| rewrite_template
| ETC ) EOB ) ;

rewrite_tree :
 ^( TREE_BEGIN rewrite_atom ( rewrite_element )* ) ;

rewrite_atom :
 ( RULE_REF
| ( ^( TOKEN_REF (a1= ARG_ACTION )? )
| CHAR_LITERAL
| STRING_LITERAL )
| LABEL
|a2= ACTION );

rewrite_template :
 ^( TEMPLATE (id= ID
|ind= ACTION ) ^( ARGLIST ( ^( ARG arg= ID a= ACTION ) )* ) (dq=
   DOUBLE_QUOTE_STRING_LITERAL
|da= DOUBLE_ANGLE_STRING_LITERAL )? ) ;

element :
 ( ^( ROOT element ) {out("^");
}

| ^( BANG element ) {out("!");
}

| atom {for (JavaAnnotationNodesParser.AnnotationContent c :
   JavaAnnotationNodesParser.table){
 if (c.method.equals(processingProduction)){
 if (c.annTable.get(ann.toString()) != null) {
 if (c.annTable.get(ann.toString()).charAt(0) == '[')
 out(c.annTable.get(ann.toString()));
 else
 out("{" + c.annTable.get(ann.toString()) + "}\n");
 }
 }
 }
}
```

```
| ^( NOT {out("~");
}
 element )
| ^( RANGE atom {out("..");
}
 atom )
| ^( CHAR_RANGE atom {out("..");
}
 atom )
| ^( ASSIGN id= ID {out(id.getText()+"=");
}
el= element )
| ^( PLUS_ASSIGN id2= ID {out(id2.getText()+"+=");
}
e2= element )
| ebnf
| tree_
|ann= ANN_COMMENT {for (JavaAnnotationNodesParser.AnnotationContent c :
   JavaAnnotationNodesParser.table){
 if (c.method.equals(processingProduction)){
 if (c.annTable.get(ann.toString()) != null) {
 if (c.annTable.get(ann.toString()).charAt(0) == '[')
 out(c.annTable.get(ann.toString()));
 else
 out("{" + c.annTable.get(ann.toString()) + "}\n");
 }
 }
 }
}

| ^( SYNPRED block [true]
) {out("=>");
}

|a= ACTION {out("{");
out(a.getText());
out("}");
}

|a2= FORCED_ACTION {out("{{");
out(a2.getText());
out("}}");
}

|pred= SEMPRED {if ( showActions )
    {
      out("{");
      out(pred.getText());
      out("}?");
    }
    else
    {
```

```
      out("{...}?");
    }
}

|spred= SYN_SEMPRED {String name = spred.getText();
GrammarAST predAST=grammar.getSyntacticPredicate(name);
block(predAST, true);
out("=>");
}

| ^( BACKTRACK_SEMPRED ( . )* )
|gpred= GATED_SEMPRED {if ( showActions ) {out("{");
   out(gpred.getText()); out("}?␣=>");}
    else {out("{...}?␣=>");}
}

| EPSILON );

ebnf :
 ( ( dotLoop )=> dotLoop
| block [true]
{out("␣");
}

| ^( OPTIONAL block [true]
) {out("?␣");
}

| ^( CLOSURE block [true]
) {out("*␣");
}

| ^( POSITIVE_CLOSURE block [true]
) {out("+␣");
}
);

dotLoop :
 ( ^( CLOSURE dotBlock )
| ^( POSITIVE_CLOSURE dotBlock ) ) ;

dotBlock :
 ^( BLOCK ^( ALT WILDCARD EOA ) EOB ) ;

tree_ :
 ^( TREE_BEGIN element ( element )* ) ;

atom
@init {
out("␣");
}
 :
 ( ( ^(rr= RULE_REF {out(($start).toString());
```

```
}
 (rarg= ARG_ACTION {out("["+rarg.toString()+"]");
}
)? ( ast_suffix )? )
| ^(t= TOKEN_REF {out(($start).toString());
}
 (targ= ARG_ACTION {out("["+targ.toString()+"]");
}
)? ( ast_suffix )? )
| ^(c= CHAR_LITERAL {out(($start).toString());
}
 ( ast_suffix )? )
| ^(s= STRING_LITERAL {out(($start).toString());
}
 ( ast_suffix )? )
| ^( WILDCARD {out(($start).toString());
}
 ( ast_suffix )? ) ) {out("␣");
}

|l= LABEL {out("␣$"+l.getText());
}

| ^( DOT id= ID {out(id.getText()+".");
}
 atom ) );

ast_suffix :
 ( ROOT {out("^");
}

| BANG {out("!");
}
);
```

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools with Gradiance*. Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[2] John Aycock and R. Nigel Horspool. Faster generalized lr parsing. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, pages 32–46, London, UK, UK, 1999. Springer-Verlag.

[3] Jean Bovet and Terence Parr. Antlrworks: an antlr grammar development environment. *Softw. Pract. Exper.*, 38(12):1305–1332, October 2008.

[4] Andrey Breslav. Grammatic – a tool for grammar definition reuse and modularity. *Computing Research Repository (CoRR)*, 2009:Article No. 0901.2461, 2009.

[5] Tony Clark. First class grammars for language oriented programming. http://eprints.mdx.ac.uk/6152/, 2009.

[6] T. Copeland. *Generating Parsers with JavaCC*. An easy-to-use guide for developers. Centennial Books, 2007.

[7] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. *onBoard*, November 2004.

[8] Eclipse - an open development platform. (http://www.eclipse.org).

[9] Bryan Ford. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, 37(9):36–47, September 2002.

[10] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 111–122. ACM, 2004.

[11] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS (26)*, pages 140–154. IEEE Computer Society, 1998.

[12] Robert Grimm. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 38–51, New York, NY, USA, 2006. ACM.

[13] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[14] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[15] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.

[16] Patrick Keegan, Ludovic Champenois, Gregory Crawley, Charlie Hunt, and Christopher Webster. *Netbeans™ ide field guide: developing desktop, web, enterprise, and mobile applications*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2005.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[18] Meta programming system. (http://www.jetbrains.com/mps/).

[19] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[20] Terence Parr. The reuse of grammars with embedded semantic actions. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 5–10, Washington, DC, USA, 2008. IEEE Computer Society.

[21] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the antlr parser generator. *SIGPLAN Not.*, 47(6):425–436, June 2011.

[22] Damijan Rebernak, Marjan Mernik, Pedro Rangel Henriques, and Maria João Varanda Pereira. Aspectlisa: An aspect-oriented compiler construction system based on attribute grammars. *Electr. Notes Theor. Comput. Sci.*, 164(2):37–53, 2006.

[23] C. Simonyi. Intentional programming overview. http://research.microsoft.com/ip/overview/, 1996.

[24] Étienne Gagnon and Etienne Gagnon. Sablecc, an object-oriented compiler framework. In *In Proceedings of TOOLS*, pages 140–154, 1998.

[25] Mark G. J. van den Brand and Eelco Visser. The Asf+Sdf meta-environment documentation tools for free! In Peter D. Mosses, Mogens Nielsen, and Michael I.

Schwartzbach, editors, *TAPSOFT 95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 803–804. Springer, 1995.

[26] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. In *Rewriting Techniques and Applications (RTA 01), volume 2051 of Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

[27] Martin P. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4), 1994.

[28] David A. Watt and Ole L. Madsen. Extended attribute grammars. *The Computer Journal*, 26:142–153, 1983. See also: report 10, Comp. Sc. Department, University of Glasgow (July 1977) and report DAIMI PB-105, Comp. Sc. Department, Aarhus University (November 1979).