Grammatical Aspects for Language Descriptions

Andrey Breslav*

ITMO University
St. Petersburg, Russia

Abstract. For the purposes of tool development, computer languages are usually described using context-free grammars with annotations such as semantic actions or pretty-printing instructions. These descriptions are processed by generators which automatically build software, e.g., parsers, pretty-printers and editing support.
In many cases the annotations make grammars unreadable, and when generating code for several tools supporting the same language, one usually needs to duplicate the grammar in order to provide different annotations for different generators.
We present an approach to describing languages which improves readability of grammars and reduces the duplication. To achieve this we use Aspect-Oriented Programming principles. This approach has been implemented in an open-source tool named Grammatic. We show how it can be used to generate pretty-printers and syntax highlighters.

1 Introduction

With the growing popularity of Domain-Specific Languages, the following types of supporting tools are created more and more frequently:

- Parsers and translators;
- Pretty-printers;
- Integrated Development Environment (IDE) add-ons for syntax highlighting, code folding and outline views.

Nowadays these types of tools are usually developed with the help of generators which accept language descriptions in the form of annotated (context-free) grammars.

For example, tools such as YACC [7] and ANTLR [12] use grammars annotated with embedded semantic actions. As an illustration of this approach first consider an annotation-free grammar for arithmetic expressions (Listing 1.1). To generate a translator, one has to annotate the grammar rules with embedded semantic actions. Listing 1.2 shows the rule `expr` from Listing 1.1 annotated for ANTLR v3.

* This work was partly done while the author was a visiting PhD student at University of Tartu, under a scholarship from European Regional Development Funds through Archimedes Foundation.

```
expr : term ((PLUS | MINUS) term)* ;
term : factor ((MULT | DIV) factor)* ;
factor : INT | '(' expr ')' ;
```

Listing 1.1. Grammar for arithmetic expressions

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

Listing 1.2. Annotated grammar rule

As can be seen, the context-free grammar rule is not easily readable in Listing 1.2 because of the actions' code interfering with the grammar notation. This problem is common for annotated grammars. We will refer to it as *tangled grammars*.

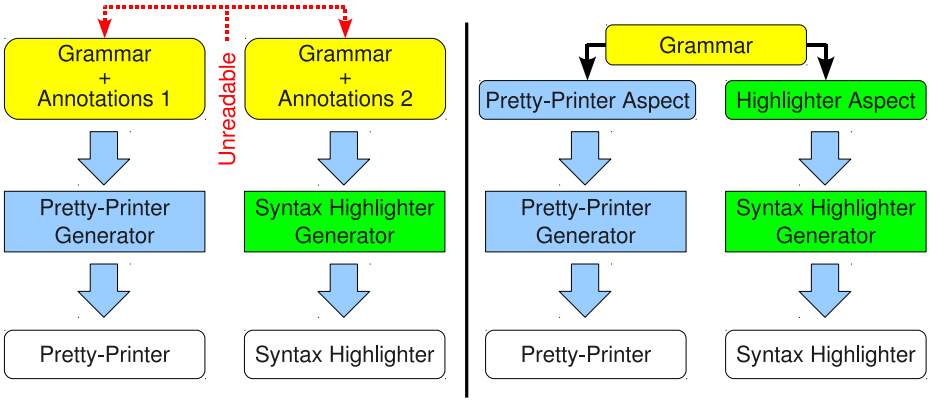

Fig. 1. Generating two supporting tools for the same language

In most applications we need to create several supporting tools for the same language (see Figure 1, left side). In such a case one uses different generators to obtain different programs (e.g., Pretzel [3] to build a pretty-printer and xText [1] to create an Eclipse editor). Each generator requires its own specific set of annotations, and the developer has to write the same grammar several times with different annotations for each generator. Besides the duplication of effort, when the language evolves, this may lead to inconsistent changes in different copies of the grammar, which may cause issues which are hard to detect. We will refer to this problem as *grammar duplication*.

This paper aims at reducing tangling and duplication in annotated grammars. A high-level view of our approach is illustrated in Figure 1 (right side): the main idea is to separate the annotations from the grammar by employing the principles similar to those behind the AspectJ language [8], this leads to a notion of a *grammatical aspect*. Our approach is implemented in an open-source tool named Grammatic[1].

In Section 2 we briefly describe the main notions of aspect-oriented programming in AspectJ. An overview of grammatical aspects and related concepts is given in Section 3. Section 4 studies the applications of Grammatic to generating syntax highlighters and pretty-printers on the basis of a common grammar. We analyze these applications and evaluate our approach in Section 5. Related work is described in Section 6. Section 7 summarises the contribution of the paper and introduces possible directions of the future work.

2 Background

Aspect-Oriented Programming (AOP) is a body of techniques aimed at increasing modularity in general-purpose programming languages by separating cross-cutting concerns. Our approach is inspired by AspectJ [8], an aspect-oriented extension of Java.

AspectJ allows a developer to extract the functionality that is scattered across different classes into modules called *aspects*. At compilation- or run-time this functionality is *weaved* back into the system. The places where code can be added are called *join points*. Typical examples of join points are a method entry point, an assignment to a field, a method call.

AspectJ uses *pointcuts* — special constructs that describe collections of join points to weave the same piece of code into many places. Pointcuts describe method and field signatures using patterns for names and types. For example, the following pointcut captures *calls of all public* `get-` *methods in the subclasses of the class* `Example` *which return* `int` *and have no arguments*:

```
pointcut getter() : call(public int Example+.get*())
```

The code snippets attached to a pointcut are called *advice*; they are weaved into every join point that matches the pointcut. For instance, the following advice writes a log record after every join point matched by the pointcut above:

```
after() : getter() {
    Log.write("A get method called");
}
```

In this example the pointcut is designated by its name, `getter`, that follows the keyword `after` which denotes the position for the code to be weaved into. An *aspect* is basically a unit comprising of a number of such pointcut-advice pairs.

[1] The tool is available at http://grammatic.googlecode.com

3 Overview of the approach

Grammatic employs the principles of AOP in order to tackle the problems of tangling and duplication in annotated grammars. We will use the grammar from Listing 1.1 and the annotated rule from Listing 1.2 to illustrate how the terms such as "pointcut" and "advice" are embodied for annotated grammars.

Grammatical join points

Figure 2 shows a structured representation (a syntax diagram) of the annotated rule from Listing 1.2. It shows the annotations attached to a symbol definition

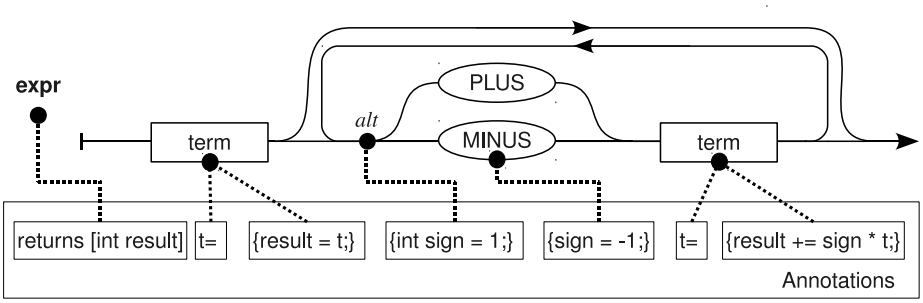

Fig. 2. Annotations attached to a grammar rule

`expr`, three symbol references: `term` (two times) and `MINUS`, and an alternative `(PLUS | MINUS)` (marked "*alt*" in the figure). All these are examples of *grammatical join points* (in Figure 2 they are marked with black circles). The full list of join point types comprises all the types of nodes of the abstract syntax trees (ASTs) of the *language of grammars*. To avoid confusion with ASTs of languages defined by the grammar, we will refer to the AST of the grammar itself as *grammar tree* (GT).

Grammatic uses a notation for grammars which is based on the one used by ANTLR. The only two differences are (i) in Grammatic productions are explicit and separated by "`:`", and (ii) an empty string is denoted explicitly by "`#empty`". Here is the list of types of GT nodes (which are also the types of the join points) with comments about the concrete syntax:

- Grammar;
- Definitions of terminal and nonterminal symbols (grammar rules) and references to them;
- Individual productions (a rule comprises one or more productions separated by "`:`");
- Concatenation (sequence), Alternative ("`|`"), Iteration ("`*`", "`+`", "`?`");
- Empty string ("`#empty`");

```
rulePattern
    : var? symbolPattern productionPattern* ';' ;
var
    : '$' NAME '=' ;
symbolPattern
    : '#'        // any symbol
    : NAME ;
productionPattern
    : var? ':' alternativePattern
    : ':' var? '{...}' ;
alternativePattern
    : sequencePattern ('|' (sequencePattern | (var? '...')))* ;
sequencePattern
    : iterationPattern+ ;
iterationPattern
    : var? atomicPattern ('*' | '+' | '?')? ;
atomicPattern
    : '(' alternativePattern ')'
    : symbolReferencePattern
    : '#empty'   // empty string
    : '..'       // any sequence
    : '#lex'     // any lexical literal
    : '$' NAME ; // a variable
```

Listing 1.3. Grammar of the pattern language

– Lexical literals (quoted strings).

The grammars given in this paper (e.g., Listing 1.3) may serve as example usages of this notation.

Grammatical pointcuts

Grammatic implements pointcuts using *patterns* over the grammar language. A pattern is an expression that matches a set of nodes in a GT. The syntax of the pattern language is given in Listing 1.3.

The most basic form of a pattern is a direct citation from a grammar:

```
expr: term ((PLUS | MINUS) term)*;
```

This pattern matches a rule of exactly the same form (rule `expr` from Listing 1.1).

In addition to this capability the pattern language makes use of various types of wildcards which make patterns more abstract and flexible. Table 1 summarizes available wildcards and the node types they each match.

Consider some examples of patterns for rules from Listing 1.1:

– `expr : {...}` — a rule defining a symbol “expr”, comprising any number of any productions (in Listing 1.1 it matches only the rule for `expr`);

Notation	Matches any...
#	Symbol
#lex	Lexical literal
..	Sequence
...	Nonempty set of alternatives
{...}	Nonempty set of productions

Table 1. Wildcards

- `# : term ..` — a production for any symbol, starting with a reference to a symbol named "term" (also matches only the rule for `expr`);
- `# : # (..)*` — a symbol reference followed by a star iterating an arbitrary sequence (matches the rules for `expr` and `term`).

The pattern language also supports variables: a part of a pattern may be associated with a name which may be used later in the same pattern, for example:

```
# : $tr=# ((PLUS | MINUS) $tr)*
```

Here the variable `$tr` is defined with the pattern # (any symbol) which means that all usages of the variable will match only occurrences of the same symbol. This pattern matches the rule for `expr` because the same symbol `term` is referenced in the positions matched by the variable `$tr`.

Note that in general a variable is bound to a *set* of GT nodes: if we match the rule for `expr` against the pattern in the example above, the variable `$tr` will be bound to a set comprised by two distinct references to the symbol `term`.

Grammatical advice

Annotations attached to grammars (they are analogous to AspectJ's *advice*) may have an arbitrarily complicated structure: in general, a generator may need a very rich annotation system. Grammatic provides a *generic annotation language*, which represents the annotations as sets of name-value pairs (see Listing 1.4) which we call *attributes*. Examples of such pairs are given in Table 2 which shows all the predefined value types. Values may also have user-defined types which can be plugged into the position marked by `<additionalValueTypes>` in the grammar.

Example	Value type
int = 10	Integer
str = 'Hello'	String
id = SomeName	Name literal
rec = {b = c; d = 5}	Annotation
seq = {{1, a b 'str'}}	Sequence of values

Table 2. Predefined value types

```
annotation
    : '{' (attribute (';' attribute?)*)? '}'
    : '.' attribute ;
namespace
    : NAME ':' ;
attribute
    : namespace? NAME ('=' value)? ;
value
    : character
    : INT
    : STRING
    : NAME
    : annotation
    : '{{' (value | punctuation)* '}}'
    : <additionalValueTypes> ;
punctuation
    : ''' | '~' | '!' | '@' | '#' | '$' | '%' | '^' | '&' | '*'
    | '(' | ')' | '-' | '+' | '=' | '|' | '\\'| '[' | ']' | ';'
    | ':' | ',' | '.' | '/' | '?' | '<' | '>' ;
```

Listing 1.4. Grammar of the advice language

For example, the annotations in Figure 2 may be represented as values of type String (other representations are also possible).

As the usage of the term "attribute" may be misleading in this context, we would like to note that the approach presented here does not directly correspond to attribute grammars [9]. In fact, grammars with annotations do not have any particular execution semantics (each generator interprets the annotations in its own way), as opposed to attributed grammars which have a fixed execution semantics. One can describe attribute grammars using Grammatic and define corresponding semantics in a generator, but this is just an example application.

Grammatical aspects

Now, having described all the components, we can assemble a *grammatical aspect* as a set of pointcuts-advice pairs. Usage of grammatical aspects is illustrated by Figure 1 (right side).

The syntax of grammatical aspects is given in Listing 1.5. An aspect consists of an optional *grammar annotation* and zero or more *annotation rules*. Annotation rules associate grammatical pointcuts (rule patterns) with advice (annotations). Here is an example of an annotation rule:

```
expr : $tr=# (.. $tr)*   // pointcut (pattern)
    $tr.varName = 't' ;  // advice (annotation)
```

In a simple case exemplified here, an annotation (`.varName = 't'`, the alternative syntax is {`varName = 't'`}) is attached to GT nodes to which a

```
aspect
    : grammarAnnotation? annotationRule* ;
grammarAnnotation
    : annotation ;
annotationRule
    : multiplicity? rulePattern subrules ;
subrules
    : (subpattern | variableAnnotation)* ;
subpattern
    : '@' multiplicity?  (productionPattern | alternativePattern)
          ':' (subrules | annotation) ;
variableAnnotation
    : '$' NAME annotation ;
multiplicity
    : '[' intOrInfinity ('..' intOrInfinity)? ']' ;
intOrInfinity
    : INT | '*' ;
```

Listing 1.5. Grammar of the aspect language

variable (`$tr`) is bound. For more complicated cases, one can define *subpatterns* — patterns which are matched against nodes situated under the matched one in the GT. For example, the following construct attaches an attribute named `varName` to each reference to the symbol `term` *inside a rule matched by a top-level pattern*:

```
expr : ..                  // pointuct (pattern)
    @$tr=(term):           // pointcut (subpattern)
       $tr.varName = 't' ; // advice (annotation)
```

This example illustrates the typical usage of subpatters where all annotations are associated with a variable bound to the whole pattern. As a shorthand for this situation Grammatic allows to omit the variable (it will be created implicitly). Using this shorthand we can abridge the previous example to the following:

```
expr : ..
    @term: { varName = 't' } ; // '{ a = b }' equals '.a = b'
```

Note that subpatterns may have their own subpatterns.

Patterns and subpatterns may be preceded by a *multiplicity directive*, for example

```
[0..1] # : $tr=# (.. $tr)* // pointcut with multiplicity
    // some advice
```

Multiplicity determines a number of times the pattern is allowed to match. The default multiplicity is `[1..*]` which means that each pattern with no explicit multiplicity is allowed to match one or more times. When an aspect is applied to a grammar, if the actual number of matches goes beyond the range allowed by a multiplicity directive, Grammatic generates an error message. In the example

above, such a message will be generated for the grammar from Listing 1.1 because the pattern matches two rules: `expr` and `term`, which violates the specified multiplicity `[0..1]`.

Generation-time behaviour

Grammatical aspects are applied at generation time. Before a generator starts working, in order to prepare the data for it, GRAMMATIC performs the following steps:

- parse the `grammar` and the `aspect`
- attach the *grammar annotation* to the root node of the `grammar`
- **for each** `annotation rule` **in** `aspect`
  - **call** APPLYPATTERN(`rule pattern`, `grammar`)

Where APPLYPATTERN is a recursive subroutine defined by the following pseudocode:

APPLYPATTERN (`pattern`, `node`) **is**

- find `subnodes` matching `pattern` among descendants of `node`
  (Variable bindings are saved in `boundTo` *map)*
- **if** the number of `subnodes` violates `pattern.multiplicity`
  - Report **error** and stop
- **for each** `subnode` **in** `subnodes`
  - **for each** `subpattern` **in** `pattern.subpatterns`
    - **call** APPLYPATTERN(`subpattern`, `subnode`)
  - **for each** `var` **in** `pattern.variables`
    - **for each** `boundNode` **in** `boundTo(var)`
      - attach `var.annotation` to `boundNode`

end

The innermost loop goes through the set of GT nodes to which the variable `var` is bound (see Section 3) and attaches the annotations associated with this variable to each of these nodes.

If no error was reported, the resulting structure (GT nodes with attached annotations) is passed to the generator which processes it as a whole and needs no information about aspects.

Thus, GRAMMATIC works as a front-end for generators that use its API. To use a pre-existing tool, for example, ANTLR, with grammatical aspects, one can employ a small generator which calls GRAMMATIC to apply aspects to grammars, and produces annotated grammars in the ANTLR format.

4 Applications

In this section we show how one can make use of grammatical aspects when generating syntax highlighters and pretty-printers on the basis of the same grammar.

```
normalClassDeclaration
    : 'class' IDENTIFIER typeParameters?
          ('extends' type)? ('implements' typeList)? classBody ;
classBody
    : '{' classBodyDeclaration* '}' ;
typeParameters
    : '<' typeParameter (',' typeParameter)* '>' ;
typeParameter
    : IDENTIFIER ('extends' bound)? ;
bound
    : type ('&' type)* ;
type
    : IDENTIFIER typeArgs? ('.' IDENTIFIER typeArgs?)* ('[' ']')*
    : basicType ;
typeArgs
    : '<' typeArgument (',' typeArgument)* '>' ;
typeArgument
    : type
    : '?' (('extends' | 'super') type)? ;
```

Listing 1.6. Class declaration syntax in Java 5

Specifying syntax highlighters

A syntax highlighter generator creates a highlighting add-on for an IDE, such as a script for `vim` editor or a plug-in for Eclipse. For all targets the same specification language is used: we annotate a grammar with *highlighting groups* which are assigned to occurrences of terminals. Each group may have its own color attributes when displayed. Common examples of highlighting groups are *keyword*, *number*, *punctuation*.

In many cases syntax highlighters use only lexical analysis, but it is also possible to employ light-weight parsers [11]. In such a case grammatical information is essential for a definition of the highlighter. Below we develop an aspect for the Java grammar which defines groups for keywords and for *declaring occurrences* of class names and type parameters. A declaring occurrence is the first occurrence of a name in the program; all the following occurrences of that name are *references*. Consider the following example:

```
class Example<A, B extends A> implements Some<? super B>
```

This illustrates how the generated syntax highlighter should work: the declaring occurrences are underlined (occurrences of `?` are always declaring) and the keywords are shown in bold. This kind of highlighting is helpful especially while developing complicated generic signatures.

Listing 1.6 shows a fragment of the Java grammar [4] which describes class declarations and type parameters. In Listing 1.7 we provide a grammatical aspect which defines three highlighting groups: *keyword*, *classDeclaration* and *typeParameterDeclaration*, for join points inside these rules.

```
# : 'class' IDENTIFIER ..
    @#lex: { group = keyword } ;
    @IDENTIFIER: { group = classDeclaration } ;
typeParameter : IDENTIFIER ..
    @#lex: { group = keyword } ;
    @IDENTIFIER: { group = typeParameterDeclaration } ;
typeArgument : {...}
    @#lex: { group = keyword } ;
    @'?': { group = typeParameterDeclaration } ;
```

Listing 1.7. Highlighting aspect for class declarations in Java

Each annotation rule from Listing 1.7 contains two subpatterns. The first one is `#lex`: it matches every lexical literal. For example, for the first rule it matches `'class'`, `'extends'` and `'implements'`; the highlighting group `keyword` is assigned to all these literals.

The second subpattern in each annotation rule is used to set a corresponding highlighting group for a declaring occurrence: for classes and type parameters it matches `IDENTIFIER` and for wildcards — the `'?'` literal.

When the aspect is applied to the grammar, GRAMMATIC attaches the `group` attribute to the GT nodes matched by the patterns in the aspect. The obtained annotated grammar is processed by a generator which produces code for a highlighter.

Specifying pretty-printers

By applying a different aspect to the same grammar (Listing 1.6), one can specify a pretty-printer for Java. A pretty-printer generator relies on annotations describing how tokens should be aligned by inserting whitespace between them.

In Listing 1.8 these annotations are given in the form of attributes `before` and `after`, which specify whitespace to be inserted into corresponding positions. Values of the attributes are *sequences* ({{ ... }}) of strings and name literals `increaseIndent` and `decreaseIndent` which control the current level of indentation.

The most widely used values of `before` and `after` are specified in a *grammar annotation* by attributes `defaultBefore` and `defaultAfter` respectively, and not specified for each token individually. In Listing 1.8 the default formatting puts nothing before each token and a space — after each token; it applies whenever no value was set explicitly.

5 Discussion

This paper aims at coping with two problems: tangled grammars and grammar duplication. When using GRAMMATIC, a single annotated grammar is replaced

```
{ // Grammar annotation
    defaultAfter = {{ ' ' }};
    defaultBefore = {{ '' }};
}

classBody : '{' classBodyDeclaration* '}'
    @'{': { after = {{ '\n' increaseIndent }} } ;
    @classBodyDeclaration: { after = {{ '\n' }} } ;
    @'}': {
        before = {{ decreaseIndent '\n' }};
        after = {{ '\n' }};
    };
typeParameters : '<' typeParameter (',' typeParameter)* '>'
    @'<': { after = {{ '' }} } ;
    @typeParameter: { after = {{ '' }} } ;
```

Listing 1.8. Pretty-printing aspect for class declarations in Java

by a *pure* context-free grammar and a set of grammatical aspects. This means that the problem of *tangled grammars* is successfully addressed.

This also means that the grammar is written down only once even when several aspects are applied (see the previous section). But if we look at the aspects, we see that the patterns carry on some extracts from the grammar thus it is not so obvious whether our approach helps against the problem of *duplication* or not. Let us examine this in more details using the examples from the previous section.

From the perspective of grammar duplication, the worst case is an aspect where all the patterns are exact citations from the grammar (no wildcards are used, see Listing 1.8). This means that a large part of the grammar is completely duplicated by those patterns. But if we compare this with the case of conventional annotated grammars, there still is at least one advantage of using Grammatic. Consider the scenario when the grammar has to be changed. In case of conventional annotated grammars, the same changes must be performed once for each instance of the grammar and there is a risk of inconsistent changes which are not reported to the user. In Grammatic, on the other hand, a developer can control this using *multiplicities*: for example, check if the patterns do not match anything in the grammar and report it (since the default multiplicities require each pattern to match at least once, this will be done automatically). Thus, even in the worst case, grammatical aspects make development less error-prone.

Using wildcards and subpatterns as it is done in Listing 1.7 (i) reduces the duplication and (ii) makes a good chance that the patterns will not need to be changed when the grammar changes. For example, consider the first annotation rule from Listing 1.7: this rule works properly for both Java version 1.4 and version 5 (see Listing 1.9 and Listing 1.6 respectively). The pointcut used in this rule is sustainable against renaming the symbol on the left-hand

```
classDeclaration
    : 'class' IDENTIFIER ('extends' type)?
                         ('implements' typeList)? classBody ;
```

Listing 1.9. Class declaration rule in Java 1.4

side (`classDeclaration` was renamed to `normalClassDeclaration`) and structural changes to the right-hand side (type parameters were introduced in Java 5). The only requirement is that the definition should start with the `'class'` keyword followed by the `IDENTIFIER`.

In AOP, the duplication of effort needed to modify pointcuts when the main program changes is referred to as the *fragile pointcut problem* [14]. Wildcards and subpatterns make pointcuts more *abstract*, in other words, they widen the range of join points matched by the pointcuts. From this point of view, wildcards help to abstract over the contents of the rule, and subpatterns — over the positions of particular elements within the rule. The more abstract a pointcut is, the less duplication it presents and the less fragile it is.

The most abstract pointcut does not introduce any duplication and is not fragile at all. Unfortunately, it is also of no use, since it matches any possible join point. This means that eliminating the duplication completely from patterns is not technically possible. Fortunately, we do not want this: if no information about a grammar is present in an aspect, this makes it much less readable because the reader has no clue about how the annotations are connected to the grammar. Thus, there is a trade-off between the readability and duplication in grammatical aspects and a developer should keep pointcuts as abstract as it is possible without damaging readability.

To summarize, our approach allows one to keep a context-free grammar completely clean by moving annotations to aspects and to avoid any unnecessary duplication by using abstract pointcuts.

6 Related work

Several attribute grammar (AG) systems, namely JastAdd [5], Silver [15] and LISA [13], successfully use aspects to attach attribute evaluation productions to context-free grammar rules. AGs are a generic language for specifying computations on ASTs. They are well-suited for tasks such as specifying translators in general, which require a lot of expressive power. But the existence of more problem-oriented tools such as Pretzel [3] suggests that the generic formalism of AGs may not be the perfect tool for problems like generating pretty-printers. In fact, to specify a pretty-printer with AGs one has to produce a lot of boilerplate code for converting an AST into a string in concrete syntax. As we have shown in Section 4, Grammatic facilitates creation of such problem-oriented tools providing the syntactical means (grammatical aspects) to avoid tangled grammars and unnecessary duplication.

The MPS [6] project (which lies outside the domain of textual languages since the editors in MPS work directly on ASTs) implements the approach which is very close to ours. It uses aspects attached to the *concept language* (which describes abstract syntax of MPS languages) to provide input data to generators. The implementation of aspects in MPS is very different from the one in Grammatic: it does not use pointcuts and performs all the checking while the aspects are created.

There is another approach to the problems we address: parser generators such as SableCC [2] and ANTLR [12] can work on annotation-free grammars and produce parsers that build ASTs automatically. In this way the problems induced by using annotations are avoided. The disadvantage of this approach is that the ASTs must be processed manually in a general-purpose programming language, which makes the development process less formal and thus more error-prone.

7 Conclusion

Annotated grammars are widely used to specify inputs for various generators which produce language support tools. In this paper we have addressed the problems of tangling and duplication in annotated grammars. Both problems affect maintainability of the grammars: tangled grammars take more effort to understand, and duplication, besides the need to make every change twice as the language evolves, may lead to inconsistent changes in different copies of the same grammar.

We have introduced *grammatical aspects* and showed how they may be used to cope with these problems by separating context-free grammars from annotations.

The primary contribution of this paper is a tool named Grammatic which implements an aspect-oriented approach to specification of annotated grammars. Grammatic provides languages for specifying grammatical pointcuts, advice and aspects.

We have demonstrated how Grammatic may be used to generate a syntax highlighter and a pretty-printer by applying two different aspects to the same grammar. We have shown that grammatical aspects help to "untangle" grammars from annotations, and eliminate the unnecessary duplication. The possible negative impact of remaining duplication (necessary to keep the aspects readable) can be addressed in two ways: (i) *abstract patterns* reduce the amount of changes in aspects per change in the grammar, and (ii) *multiplicities* help to detect inconsistencies at generation time.

One possible way to continue this work is to support grammar adaptation techniques [10] in Grammatic to facilitate rephrasing of syntax definitions (e.g., left factoring or encoding priorities of binary operations) to satisfy requirements of particular parsing algorithms.

Another possible direction is to generalize the presented approach to support not only grammars, but also other types of declarative languages used as inputs for generators, such as UML or XSD.

References

1. The Eclipse Foundation. Xtext. http://www.eclipse.org/Xtext, 2009.
2. Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140, Washington, DC, USA, 1998. IEEE Computer Society.
3. Felix Gärtner. The PretzelBook. Available online at http://www.informatik.tu-darmstadt.de/BS/Gaertner/pretzel/ (last accessed on April 28, 2010), 1998.
4. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.
5. Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
6. JetBrains. Meta Programming System (MPS). http://www.jetbrains.com/mps, 2009.
7. Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, 1979.
8. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
9. Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.
10. Ralf Lämmel. Grammar adaptation. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 2001.
11. Leon Moonen. Generating robust parsers using island grammars. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.
12. Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
13. Damijan Rebernak and Marjan Mernik. A tool for compiler construction based on aspect-oriented specifications. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 11–16, Washington, DC, USA, 2007. IEEE Computer Society.
14. Maximilian Störzer and Christian Koppen. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.
15. Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *ENTCS*, 203(2):103–116, 2008.