Grammatical Aspects: Coping with Duplication and Tangling in Language Specifications[1]

Andrey BRESLAV
ITMO University, Saint-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 specifications 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.

Keywords. DSL, Grammar, Aspect

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 specifications 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

[1]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. Author would like to acknowledge the advice received from Professor Marlon Dumas.

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

Listing 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 2: Annotated grammar rule

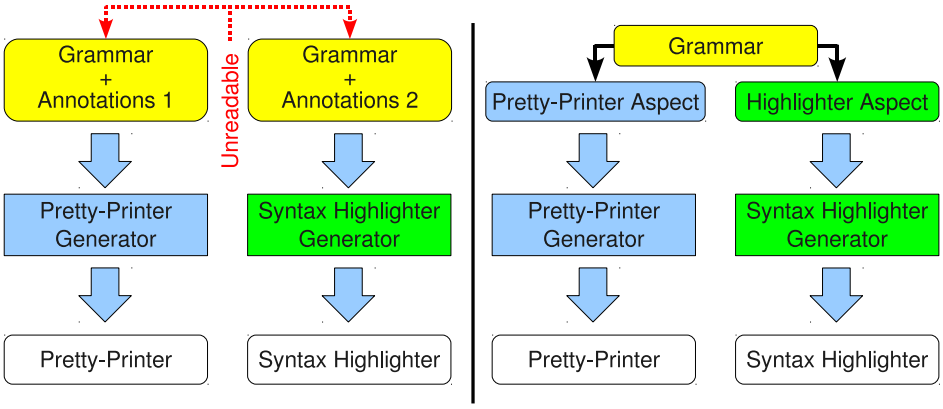

Figure 1. Generating two supporting tools for the same language

consider an annotation-free grammar for arithmetic expressions (Listing 1). To generate a translator, one has to annotate the grammar rules with embedded semantic actions. Listing 2 shows the rule `expr` from Listing 1 annotated for ANTLR v3.

As can be seen, the context-free grammar rule is not easily readable in Listing 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*.

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[2].

In Section 1 we briefly describe the main notions of aspect-oriented programming in AspectJ. Definitions related to grammatical aspects and their formal semantics are given in Section 2. Section 3 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 4. Related work is described in Section 5. Section 6 summarises the contribution of the paper and introduces possible directions of the future work.

1. 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.

2. Overview of the approach

Out approach employs the principles of AOP in order to tackle the problems of tangling and duplication in annotated grammars. We will use the grammar from

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

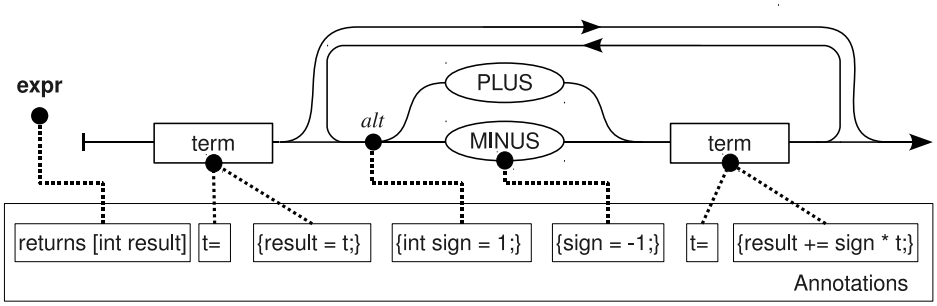

Figure 2. Annotations attached to a grammar rule

Listing 1 and the annotated rule from Listing 2 to illustrate how the terms such as "pointcut" and "advice" are embodied for annotated grammars.

2.1. Grammatical join points

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

Below we use ANTLR's notation for context-free grammars and reflect on ASTs of this notation. 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). The language of GTs is given by the following type definitions:

```
data Grammar = Grammar {rules :: Set(Rule)}
data Rule = Rule {name :: String, body :: Expression}
data Expr -- Expressions
    = SymRef {name :: String}  -- Symbol reference
    | Literal {value :: String} | Empty
    | Seq {exprs :: List(Expr)} -- Sequence
    | Alt {exprs :: Set(Expr)}  -- Alternative
    | Iter {exprs :: Expr, multiplicity :: Mult}  -- Iteration
data Mult = Star | Plus | Option
```

Every node in a GT can be a join point, thus we have as many join point types as data constructors in the GT language above.

2.2. Grammatical pointcuts

Pointcuts are expressions denoting sets of join points, i.e. *patterns* over the GT language. The language of pointcuts is obtained from the GT language by adding special constructs, namely wildcards and variable bindings (see Table 1). Consider some examples of patterns for the rules from Listing 1:

(a) `expr : [?]` — a rule defining a symbol "expr" with an arbitrary expression on the right-hand side (in Listing 1 it matches only the rule for `expr`);
(b) `? : term [?]` — a rule for any symbol, starting with a reference to a symbol named "term" (also matches only the rule for `expr`);

Notation	Matches any...
?*Type*	Any instance of type *Type*
?	Any symbol (?SymRef)
?lex	Any lexical literal (?Literal)
[?]	Any sequence
{?}	Any set of alternatives
$\lambda var \approx expr_1 \,.\, expr_2(var)$	Variable binding

Table 1. Notions introduced in the pointcut language (in addition to the GT nodes)

(c) `? : ? [?]*` — a symbol reference followed by a star iterating an arbitrary sequence (matches the rules for `expr` and `term`).

Note that constructors of the GT language are used in patterns (typing is relaxed). Here is the example (b) from above written in the abstract syntax (":" denotes the *cons* operation on lists): $Rule(?SymRef, Seq(SymRef(\text{"term"}) : ?List(Expr)))$.

Variables are used to label a subexpression and refer to it later. In the abstract syntax this can be written, for example, as

$\lambda tr \approx ?SymRef \,.\, tr\ ((\texttt{PLUS} \mid \texttt{MINUS})\ tr)^*$

Here the variable `tr` is defined with the pattern $?SymRef$ (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`. In concrete syntax variables are defined inside the pattern, and prefixed by a dollar sign so that the pattern given above is written as follows:

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

In this paper we assume that GT nodes are *not* shared, and subtrees of the same shape occurring at different positions are distinct. This corresponds to having unique numbers implicitly assigned to nodes, below we refer to this numbers as *identity labeling* and write $\mathcal{IL}(n)$ for the identity-label of a node n. In this regard, note that a variable is in general 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 `term`.

2.3. 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. We use a *generic annotation language*, which represents the annotations as sets of name-value pairs 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 in. The annotations in Figure 2 may be represented, for example, 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 attribute grammars which have a fixed execution se-

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

mantics. One can describe attribute grammars using our approach and define corresponding semantics in a generator, but this is just an example application.

2.4. Grammatical aspects

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

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'}` is attached to GT nodes to which a variable `tr` is bound. For more complicated cases, one can define arbitrarily nested *subpatterns* — patterns being matched against nodes situated in a particular GT subtree. 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 subpatterns where all annotations are associated with a variable bound to the whole pattern. As a shorthand for this situation the variable can be omitted (it will be created implicitly). Using this shorthand we can abridge the previous example to the following:

```
expr : [?]
    @term: { varName = 't' } ;
```

2.5. Semantics of pointcuts and aspects

To precisely define what happens when an aspect is applied to a language specification, in this section we give formal semantics to the constructs described above. The key point in this formalization is to characterize pointcut matching by establishing a system of inequalities over sets of terms[3] corresponding to pointcuts. Solutions of these inequalities give valuations to variables defined inside patterns and subpatterns.

[3]Since terms are trees, in this section we use "GT node" and "term" interchangeably.

$$
\begin{aligned}
\mathcal{S}(V, \mathbf{Type}(t_1, \ldots, t_n)) &= \{V \in \mathbf{Type}(v_1^\dagger, \ldots, v_n^\dagger)\} \cup \textstyle\bigcup_1^n \mathcal{S}(v_i, t_i), \\
&\text{where } \mathbf{Type} \in \{\texttt{Grammar}, \texttt{Rule}, \texttt{Iter}, \texttt{SymRef}, \texttt{Literal}, \texttt{Empty}\} \\
\mathcal{S}(V, ?Type) &= \{V \in [\![Type]\!]\} \\
\mathcal{S}(V, \lambda\, v \approx e_1\,.\,e_2) &= \mathcal{S}(v, e_1) \cup \mathcal{S}(V, e_2) \\
\mathcal{S}(V, Seq(e_1, \ldots, e_n)) &= \{V \in Seq(L^\dagger)\} \cup \{L = L_1^\dagger \oplus \ldots \oplus L_n^\dagger\} \cup \textstyle\bigcup_1^n \mathcal{S}(L_i, e_i) \\
\mathcal{S}(V, [?]) &= \{V \in [\![List(Expr)]\!]\} \\
\mathcal{S}(V, Alt(e_1, \ldots, e_n, \{?\})) &= \{V \in Alt(S^\dagger)\} \cup \textstyle\bigcup_1^n \mathcal{S}(t_i^\dagger, e_i) \cup \\
&\quad \cup \textstyle\bigcup_1^n \{t_i \in S\} \cup \{R^\dagger = S \setminus \bigcup_1^n \{t_i\}\} \\
\mathcal{S}(V, v) &= \{V = v\}
\end{aligned}
$$

Figure 3. Transforming a pattern into a system of inequalities

Definition 1 (System of set inequalities)**.** *Given an infinite set of variables $\mathcal{V}$, finite set of types $\mathcal{T}$ and value constructors $\mathcal{C}$ (we denote the set of all possible constructor applications as $\overline{\mathcal{C}}$), a* system of set inequalities *is a finite set of expressions of the following forms:*

- *$x \in S$, where $x \in \mathcal{V} \cup \overline{\mathcal{C}}$,*
- *or $S_1 = S_2$, where $S_i \in \mathcal{V} \cup \overline{\mathcal{C}}$ or $S_i \subseteq \overline{\mathcal{C}}$.*

Given such a system S, a function $\sigma : \mathcal{V} \to \overline{\mathcal{C}}$ is called a solution *of S, iff replacing every occurrence of every $v \in \mathcal{V}$ in S by $\sigma(v)$ turns all the entries of S into true statements.*

In addition to the common set operations, we use the following conventions:

- $[\![T]\!]$, where $T \in \mathcal{T}$ denotes a set of all values of type T.
- If $C \in \mathcal{C}$, then $C(S_1, \ldots, S_n) = \{C(s_1, \ldots, s_n) \mid s_1 \in S_1, \ldots, s_n \in S_n\}$.
- For $t \in \overline{\mathcal{C}}$, $Subterms(t)$ is the set of all direct and indirect subterms of t (identity-labeled).
- $L_1 \oplus L_2 = \{l_1 \oplus l_2 \mid l_1 \in L_1, l_2 \in L_2\}$, where $\oplus$ denotes list concatenation.
- A single value can be treated as a singleton set or singleton list containing this value, if required by the context.

Figure 3 defines a function $\mathcal{S}$ that takes a variable and a pattern as arguments and returns a system of set inequalities. We denote variables v which are supposed to be fresh with respect to the rest of the system as $v^\dagger$.

Definition 2 (Matching, Root variable)**.** *Given a pointcut expression P and a GT t, let*

$$S_P(t) = \mathcal{S}(R^\dagger, P) \cup \{R = t\}$$

R is called the root variable*. Then, P* matches *t iff there is a solution σ to $S_P(t)$.*

To incorporate a subpattern P_1 into this definition, it is enough to consider an extended system $S_{P,P_1}(t) = S_P(t) \cup S_{P_1}(t) \cup \{R_1 \in Subterms(R)\}$, where R is the root variable for $S_P(t)$, and R_1 is the root variable for $S_{P_1}(t)$. Arbitrarily nesting subpatterns are easily expressible in this manner.

According to these definitions, if P matches t, then if we substitute all variables in $S_P(t)$ by their values given by σ, all the entries in the system turn into

true statements. This gives a precise definition of matching, but does not yet define what values are bound to variables defined in P (we refer to the set of such variables as $\mathcal{V}_P$). Note that according to Figure 3, every $v \in \mathcal{V}_P$ appears in $S_P(t)$, and thus, $\sigma(v)$ seems to be a good candidate, but it is not: it only gives a term *structurally equivalent* to all values bound by v, but not the set of actual identity labels for particular subterms. However, this set can be reconstructed during the substitution of $\sigma(v)$ into $S_P(t)$: whenever we substitute a term for $v \in \mathcal{V}_P$, we label this term with v (a single term may be labeled by many variables). In the end, we substitute a term t' for the root variable R, and it is structurally equivalent to t as stated by the Definition 2. Given the identity labels in t and variable labels in t', we can derive a function called σ*-labeling* of t:

$$\mathcal{L}_\sigma : \mathcal{IL}(Subterms(t)) \to 2^{\mathcal{V}},$$

where $\mathcal{L}_\sigma(n)$ is the labeling of the corresponding term in t'. Using this function, we can define the set of variables bounded to a variable v:

Definition 3 (Bounding function)**.** *A function* $\mathcal{B}_\sigma : \mathcal{V}_P \to 2^{\mathcal{IL}(Subterms(t))}$*, such that*

$$\mathcal{B}_\sigma(v) = \{n \mid v \in \mathcal{L}_\sigma(n)\},$$

is called a bounding function *for* $S_P(t)$*.*

To complete the semantics of aspects, note that an aspect rule r gives a function $V_r(v)$ returning an annotation to be associated to every $n \in \mathcal{B}_\sigma(v)$. To apply a particular rule r (a pointcut P_r and the annotation function V_r), for every node t in the GT, if there is a solution σ for $S_{P_r}(t)$, then for every $v \in \mathcal{V}_{P_r}$ associate $V_r(v)$ with every $n \in \mathcal{B}_\sigma(v)$. If there are two annotations associated to the same node, they are merged (duplicate attributes, if any, are kept). Naturally, the grammar annotation (if present) is simply associated to the whole grammar.

2.6. Implementation

Our approach is implemented as an open-source tool Grammatic[4] written in Java. The tool provides concrete syntax for grammar specifications and aspects and implements the matching and application semantics given above.

Grammatic works as a front-end to generators that use its API. First, it takes a specification and applies aspects. If the application was successful, 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. 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.

From the practical point of view, it is important that the definitions in the previous section allow a single join point to match more than once. Depending

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

on the grammar, this behavior may be intended by the developer, or not. To provide some means of control over this behavior, patterns and subpatterns may be preceded by a *multiplicity directive*. For example:

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

A multiplicity directive 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, an error message will be generated. In the example above, there will be an error when matching against the grammar from Listing 1 because the pattern matches two rules: `expr` and `term`, which violates the specified multiplicity `(0..1)`.

3. 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.

3.1. 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 3 shows a fragment of the Java grammar [4] which describes class declarations and type parameters. In Listing 4 we provide a grammatical aspect which defines three highlighting groups: *keyword*, *classDeclaration* and *typeParameterDeclaration*, for join points inside these rules.

Each annotation rule from Listing 4 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.

```
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 3: Class declaration syntax in Java 5

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

Listing 4: Highlighting aspect for class declarations in Java

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.

3.2. Specifying pretty-printers

By applying a different aspect to the same grammar (Listing 3), 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 5 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 liter-

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

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

Listing 5: Pretty-printing aspect for class declarations in Java

als `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 5 the default formatting puts nothing before each token and a space — after each token; it applies whenever no value was set explicitly.

4. Discussion

This paper aims at coping with two problems: tangled grammars and grammar duplication. When using Grammatic, a single annotated grammar is replaced 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 5). 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

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

Listing 6: Class declaration rule in Java 1.4

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 4 (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 4: this rule works properly for both Java version 1.4 and version 5 (see Listing 6 and Listing 3 respectively). The pointcut used in this rule is sustainable against renaming the symbol on the left-hand 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.

5. 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 3, 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.

6. 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 contributions of this paper are:

- Formal definition of grammatical aspects and their semantics.
- A tool named Grammatic which provides languages for specifying grammatical pointcuts, advice and aspects, and implements the semantics of matching and aspect application.

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.