$\mathcal{X}$AGra - An XML dialect for Attribute Grammars

Nuno Oliveira[1], Pedro Rangel Henriques[1], Daniela da Cruz[1], and Maria João Varanda Pereira[2]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{nunooliveira,prh,danieladacruz}@di.uminho.pt
[2] Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt

Abstract. Attribute Grammars (AG) are a powerful and well-known formalism used to create language processors. The meta-language used to write an AG (to specify a language and its processor) depends on the compiler generator tool chosen. This fact can be an handicap when it is necessary to share or transfer information between language-based systems; this is, we face an interchangeability problem, if we want to reuse the same language specification (the AG) on another development environment.
To overcome this interoperability flaw, we present in this paper $\mathcal{X}$AGra - an XML dialect to describe attribute grammars. $\mathcal{X}$AGra was precisely conceived aiming at adapting the output of a visual attribute grammar editor (named Visual$\mathcal{LISA}$) to any compiler generator tool.
Based on the formal definition of Attribute Grammar and on the usual requirements for the generation of a language processor, $\mathcal{X}$AGra schema is divided into five main fragments: symbol declarations, attribute declarations, semantic productions (including attribute evaluation rules, contextual conditions, and translation rules), import, and auxiliary functions definitions. In the paper we present those components, but the focus will be on the systematic way we followed to design the XML schema based on the formal definition of AG.
To strength the usefulness of $\mathcal{X}$AGra as a universal AG specification, we show at a glance $\mathcal{X}$AGraAl, a tool taking as input an AG written in $\mathcal{X}$AGra, is a *Grammar Analyzer and Transformation system* that computes dependencies among symbols, various metrics, slices and rebuilds the grammar.

1 Introduction

In the area of language processing it is common to use *Attribute Grammars* (AG) [1] as a formalism to specify the language syntax, semantics and also the translation to target code. The notation of this kind of formalism strongly depends on the compiler generator tool (CG) we choose to automatically produce the processor. This implies the conversion of notations, in order to readapt the specification to other tools, which is a situation that occurs often.
The focus of this paper is on an XML dialect to represent universally attribute grammars, this is, in a tool-independent manner. The main idea is to generalize the output of AG

editing tools; instead of generating a description for a specific compiler generator, the editor-like tool under development can produce this general purpose dialect. Then to use this editor as a *Front End* (FE) for a specific generator, it is only necessary to resort a simple translatorto convert the XML description into the specific meta-grammar of that CG. This approach raises the usefulness of the editor-like tool, as it can be used as a FE for a larger range of grammar-based generators.
Actually, $\mathcal{X}$AGra — *an XML dialect for Attribute Grammars* — appeared during the development of Visual$\mathcal{LISA}$ [2], a visual editor for $\mathcal{LISA}$ [3] attribute grammars that enables the drawing of syntax trees (grammar productions) decorated with attributes and the respective evaluation rules to define visually a complete attribute grammar. In its original version, Visual$\mathcal{LISA}$ converts this visual description into $\mathcal{LISA}$ meta-language. To extend the use of that visual environment for AG specification, in order to cope with different compiler generator toolswas built to translate the visual representation into $\mathcal{X}$AGra.
This XML dialect allows the declaration of all grammar symbols (terminals and non-terminals) and all (inherited and synthesized) attributes, and also the definition of all semantic rules (to evaluate the attributes, define contextual conditions, and translate the language), which are the elements needed to specify a complete attribute grammar. In addition, it is possible to identify modules or libraries to be imported, and also to define auxiliary functions. It is worth notice that the XML-Schema (XSD) underlying $\mathcal{X}$AGra was developed rigourously taking into account the formal definition of AG and the extra information that a CG needs to produce a complete language processor — that working approach led to a fast design and implementation.
We also envisage a broader field of applications for $\mathcal{X}$AGra representation than just the role of a general interface between editor-like tools and compiler generators. To corroborate that idea, we develop in the context of a language engineering course a powerful tool (called $\mathcal{X}$AGraAl) for AG analysis and transformation; $\mathcal{X}$AGraAl takes as input a $\mathcal{X}$AGra AG.
Due to its role in the conception of $\mathcal{X}$AGra, attribute grammars are formally defined in section 2. Then, section 3 is devoted to the introduction of $\mathcal{X}$AGra dialect (highlighting its derivation from the AG definition). Section 4 introduces $\mathcal{X}$AGraAl tool and summarizes its functionality. Conclusion and future work sum up the paper in section 5.

2 Attribute Grammars

A *Context-Free Grammar* (CFG) is formally defined by the following tuple:

$$G = (T, N, S, P)$$

where:
T is the set of terminal symbols that define the alphabet for the language;
N is the set of nonterminal symbols;
$S \in N$ is the start symbol of the grammar and
P is a set of productions.
A production, also known as derivation rule, is composed of a *Left-Hand Side* (LHS), with LHS $\in N$; and a *Right-Hand Side* (RHS), with RHS $\subseteq (N \cup T)^*$.

An Attribute Grammar is based on a `CFG`. It associates: a set, $A(X)$, of attributes with each symbol X in the vocabulary ($V = T \cup N$) of G; a set, $R(p)$, of evaluation rules with each production $p \in P$; and a set, $C(p)$, of contextual conditions with each production $p \in P$.
So an attribute grammar is formally defined as the following tuple:

$$AG = (G, A, R, C, \mathcal{T})$$

where
$A = \bigcup_{X \in (N \cup T)} A(X)$ is the set of all the attributes;
$R = \bigcup_{p \in P} R(p)$ is the set of evaluation rules for all the productions;
$C = \bigcup_{p \in P} C(p)$ is the set of contextual conditions for all the productions and
$\mathcal{T} = \bigcup_{p \in P} \mathcal{T}(p)$ is the set of translation rules for all the productions.
Each attribute has a type, and represents a specific property of symbol X; we write $X.a$ to indicate that attribute a is an element of $A(X)$. For each $X \in (N \cup T)$, the set of attributes of X is splitted into two disjoint sets: $A(X) = Inh(X) \cup Syn(X)$, respectively the *inherited* and the *synthesized* attributes.
Each $R(p)$ is a set of formulas

$$X.a = func(..., Y.b, ...)$$

that define how to compute, in the precise context of production p, the value of each attribute a as a function of the value of other attributes b, where each defined attribute a should be a synthesized attribute associated with the nonterminal in the lefthand side or an inherited attribute associated with a nonterminal in the righthand side:

$$a \in (Syn(X_0) \cup Inh(X_i)), i \geq 1$$

and each used attribute b should be an inherited attribute associated with the nonterminal in the lefthand side or a synthesized attribute associated with a symbol in the righthand side:

$$b \in (Inh(X_0) \cup Syn(X_i)), i \geq 1$$

Each $C(p)$ is a set of predicates

$$pred(..., X.a, ...)$$

describing the requirements that must be satisfied in the precise context of production p. Each predicate, checked for the actual value of the argument attributes (any synthesized or inherited attribute that occurs in that context can be an argument), must hold a `true` value, so that the production is meaningful (is valid from a semantic point of view).
Each $\mathcal{T}(p)$ is a set of procedures

$$proc(..., X.a, X.b, ...)$$

that use the value of attributes available in the context of production p (preferably the synthesized attributes of production, but not restricted to) to produce, or generate, the output of the language processor.

Many times, and many authors, do not consider $C(p)$ and $\mathcal{T}(p)$ separate from $R(p)$: they consider an AG as a triplet

$$AG = (G, A, R)$$

In these cases, contextual conditions and translation rules are defined as boolean functions that associate a truth value with special boolean attributes and produce the desired action (contextual constraint verification or output building) as a side effect.

In summary, AGs are a formal and practical way to develop any kind of programming language. The possibility to use attributes to store and spread information through the processing phase, makes easier the derivation of all modules needed to compile a language [4], and hence, it is faster to get the desired output.

3 $\mathcal{X}$AGra language

In this section is defined an XML dialect to cope with attribute grammars. We gave it the name of $\mathcal{X}$AGra, which stands for ***X**ML dialect for **A**ttribute **Gra**mmars*. From here on, this XML notation will be referred to as $\mathcal{X}$AGra.
$\mathcal{X}$AGra denotes the abstract representation of an AG. Its notation, here defined, is mainly based on the definition of AG presented in Section 2, but it also borrows parts from the notations inherent to various AG-based compiler generator tools.
One of the standardized ways to define a new XML dialect is the creation of a schema, using the standard XML Schema Definition (XSD) language. For the sake of space, the integral textual definition of $\mathcal{X}$AGra's schema is not presented, and for reasons of visibility and readability, the complete drawing of the schema is broken into several important sub-parts. These sub-parts are explained in the present section. Figures 1 to 7 are used to support the explanation of the dialect.

$\mathcal{X}$AGra's root element was defined as attributeGrammar. This element has a single attribute, name, whose objective is to store the name of the grammar, or the language that the grammar defines; and is a sequence of several elements. These elements represent components of the formal definition of an AG, incremented with extra parts related to the usage of AG-based compiler generators.
Table 1 defines a relation of inclusion between the $\mathcal{X}$AGra notation elements and the components that constitute the formal definition of an AG, which is recovered next:

$$AG = (T, N, S, P, A, R, C, \mathcal{T})$$

The relations depicted in Table 1 give an overview about the information that each element of $\mathcal{X}$AGra notation will store. The following sections will describe with more detail such elements and the information they store.
Listing 1.1 presents a fragment of a grammar that computes the age of a set of students. This example is used to compare the concrete notation of a compiler generator to the XML fragments that are shown in the sequent figures.

Table 1. Derivation of $\mathcal{X}$AGra Notation From the Formal Definition of AG

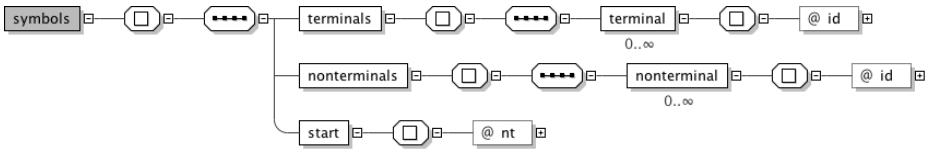
$\mathcal{X}$AGra Element $\supseteq$ AG Components	
symbols	T, N, S
attributesDecl	A
semanticProds	$P, R, C, \mathcal{T}$
importations	$\emptyset$
functions	$\emptyset$

Listing 1.1. Example of Students Grammar

```
language StudentsGra {
   lexicon{
      Name      [A-Z][a-z]+
   ...
   }
   attributes
      int   STUDENTS.sum;
   ...
   rule Students_1 {
      STUDENTS ::=  STUDENT STUDENTS compute {
         STUDENTS.sum = STUDENTS[1].sum + STUDENT.age;
      };
   }
   ...
   method user_Definitions {
      import java.util.ArrayList
      public int sum(int x, int y){
        return x+y;
      }
   }
}
```

3.1 Element *symbols*

Figure 1 presents the schema for the element `symbols`. As the name suggests, this element contains the declaration of the grammar's vocabulary.
It is composed of a sequence of three elements: `terminals`, `nonterminals` and `start`.
The element `terminals` is a sequence of zero or more elements named `terminal`, which, in its turn, has one attribute, `id`, used to store the name of a terminal symbol. This attribute is an identifier, hence any instance of it, must be different from the others, and must be always instantiated. Besides the information kept on the attribute, this element has a textual content where the respective *Regular Expression* (`RE`) can be declared.
The element `nonterminals` has similar structure. The difference lays on the fact that it represents a sequence of zero or more elements `nonterminal` which have no textual content. The attribute `id` has the same purpose as the attribute with the same name in the element `terminal`.
Finally, the element `start` has a single attribute named `nt`. This attribute is used to refer the nonterminal (already defined in the $\mathcal{X}$AGra specification), correspondent to the start symbol (or Axiom) of the AG.

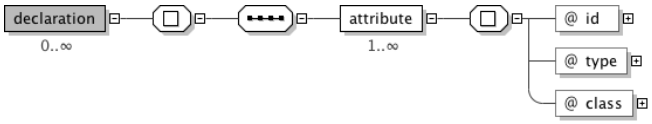



```
<symbols>
    <terminals>
        <terminal id="name">[A–Z][a–z]+</terminal>
    </terminals>
    <nonterminals>
        <nonterminal id="students" />
    </nonterminals>
    <start nt="students" />
</symbols>
```

Fig. 1. $\mathcal{X}$`AGra` Schema – Element **Symbols**: definition and example

3.2 Element *attributesDecl*

This element is composed of a sequence of zero or more elements `declaration`. For the sake of readability, Figure 2 only depicts the structure of the element `declaration`, which is a sequence of one or more elements `attribute`. This one has three mandatory attributes: i) `id` – stores the name of the attribute being declared. Any kind of text can be used to define it, but it is always better to use the following notation: $X.a$, where X is the name of a symbol in $T \cup N$ and a is the name of an attribute in $A(X)$. As it is an identifier, it must be different from all other identifiers on the specification; ii) `type` – stores the data type of the current attribute value and iii) `class` – defines the class of the attribute. It must be one of: InhAttribute, SyntAttribute and IntrinsicValueAttribute.




```
<attributesDecl>
    <declaration>
        <attribute id="students.sum" type="int" class="SyntAttribute" />
    </declaration>
</attributesDecl>
```

Fig. 2. $\mathcal{X}$`AGra` Schema – Element **Attribute Declarations**: definition and example

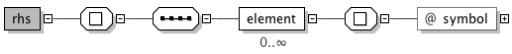
3.3 Element *semanticProds*

The element `semanticProds` represents the structure to define productions and associated semantic rules in $\mathcal{X}$`AGra` specifications. This structure is composed of a sequence of zero or more elements `semanticProd`. Each `semanticProd` has one single attribute, `name`, used to store the mandatory name of the production, as an identifier.
Element `semanticProd` has three direct descendants: `lhs`, `rhs`, `computation`, whose structure is explained in the next paragraphs and that are depicted in Figures 3, 4 and 5.
Element `lhs` (Figure 3) is used to refer to the nonterminal symbol on the `LHS` of the production. This element has a single attribute, `nt`, to refer to an existent `nonterminal`.


```
1 <lhs nt="students" />
```

Fig. 3. $\mathcal{X}$`AGra` Schema – Element **Semantic Productions**: `LHS` definition and example

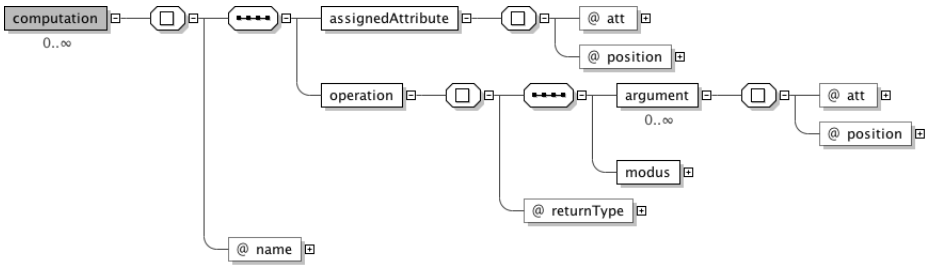
Element `rhs` (Figure 4), stores the nonterminals on the `RHS` of a production. It is composed of a sequence of zero or more elements `element`. For this purpose, each `element`, has a single attribute, `symbol`, which is mandatory and represents a reference to a terminal or nonterminal symbol, already instantiated in the initial `symbols` structure.




```
1 <rhs>
2     <element symbol="student" />
3     <element symbol="students" />
4 </rhs>
```

Fig. 4. $\mathcal{X}$`AGra` Schema – Element **Semantic Productions**: `RHS` definition and example

Element `computation` (Figure 5) is the last child of the element `semanticProds`. It represents an hard concept of `AG`s: the semantic rules.
This element has one attribute, `name`, used to give a name to the computation being declared. This attribute, despite being mandatory, is not a unique identifier: different computations can have equal names.
The structure of `computation` represents a pure abstraction of what is a semantic rule in an `AG` definition: the attribute to which a value is assigned, and the operation that




```
1 <computation name="getTheSum">
2     <assignedAttribute att="students.sum" position="0" />
3     <operation returnType="int">
4         <argument att="student.age" position="1" />
5         <argument att="students.sum" position="2" />
6         <modus> $1 + $2 </modus>
7     </operation>
8 </computation>
```

Fig. 5. $\mathcal{X}$AGra Schema – Element **Semantic Productions**: `Computation` definition and example

computes this value. Thus, the element `computation` has two children: the elements `assignedAttribute` and `operation`.

Element `assignedAttribute` is composed of two mandatory attributes: `att`, which is used to refer to an attribute; and `position`, which is a number that identifies the position of the symbol associated to the attribute in the list of elements of the production. That is, if the attribute is connected to the `LHS`, then the value for `position` must be 0. If the associated symbol belongs to the `RHS`, then its value should correspond to the position that the symbol occupies in the `RHS` sequence of symbols, starting with 1.

The element `operation` aggregates a sequence of zero or more elements `argument` and a single element `modus`. In addition to the elements, it has an attribute, `returnType`, used to store the data type of the value returned by the operation.

Elements `argument` are, in all aspects, equal to the `assignedAttribute` element. Each one has two attributes with the same name and the same semantic value underlaying, therefore they are used to refer to previous declared attributes. The difference is on the fact that this time, the attributes referenced are those used to compute the value in the operation.

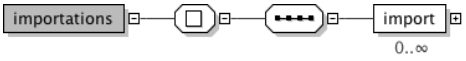
The last element, `modus`[3], which is a simple text field to write the expression used to compute the value. Somehow, in this element's text, a reference to the argument attributes should be made. An example (and the convention established) is using $\$x$, where $x > 0$ is the position of the attribute in the sequence of arguments.

The next two simple parts extend the mathematical definition of `AG`s to the abstract language of any compiler generator based on `AG`s.

[3] *modus* is a latin expression for *way* (of computing something, in our case)

3.4 Element *importations*

Figure 6 presents the structure to declare the importation of packages or programming language modules that can be necessary for the computation of all attributes. The element `importations` is a sequence of zero or more elements `import`. Each of these elements `import` is a simple text container, where the name of the package or module should be written.




```
1 <importations>
2     <import>
3         java.util.ArrayList
4     </import>
5 </importations>
```

Fig. 6. $\mathcal{X}$`AGra` Schema – Element **Importations**: definition and example

3.5 Element *functions*

In a very similar way, element `functions` is a sequence of zero or more `function` elements. Each `function` element has a mandatory attribute, `name`, used to store the name of the function. This element is defined as a text container, in order to be possible the definition of a concrete function. The code of the function must be written in the target programming language like `Java` or other.


```
1 <functions>
2     <function name="sum">
3         public int sum(int x, int y){
4           return x + y;
5         }
6     </function>
7 </functions>
```

Fig. 7. $\mathcal{X}$`AGra` Schema – Element **Functions**: definition and example

$\mathcal{X}$`AGra`'s schema is now completely defined and explained, revealing the universality needed to store any `AG` for any `AG`-based compiler generator.

4 Application Example

In this section we give a brief introduction to $\mathcal{X}$AGraAl, a *Grammar Analyzer and Transformation tool* that computes dependencies among symbols, some grammar metrics, and grammar slices for a given criterion; moreover, $\mathcal{X}$AGraAl can also derive, from the original, new shorter grammars combining slices or removing useless productions (similar to re-factoring a program). $\mathcal{X}$AGraAl takes as input an AG written in $\mathcal{X}$AGra; thus, the presentation of this tool is precisely aiming at illustrating the applicability of $\mathcal{X}$AGra as a universal AG specification language.
$\mathcal{X}$AGraAl is a platform independent tool, developed using Java. To parse the input it is used the Java Architecture for XML Binding (JAXB) [5] and Java API for XML Processing (JAXP) [6]. JAXB simplifies the access to the XML document from a Java program by presenting the document in Java format. All JAXB implementations provide a tool called a *binding compiler* to bind a schema (the way the binding compiler is invoked can be implementation-specific). *Binding a schema*, the first step in this processing approach, means generating a set of Java classes that represents the schema. Those classes are then instantiated during the parsing.
While parsing a $\mathcal{X}$AGra grammar using JAXB, $\mathcal{X}$AGraAl builds the identifiers table (IdTab) where it collects all grammar symbols and attributes; each identifier is associated with all its characteristics extracted or inferred from the source document. The identifiers table — that can be pretty-printed in HTML — complemented by the dependence graph (DG) — also printable using Dot and GraphViz — constitute the core of the tool. Traversing those internal representation structures, it is possible to implement the other $\mathcal{X}$AGraAl functionalities:

- Metrics, to assess grammar quality;
- Slicing, to ease the analysis producing sub-grammars focussed in a specific symbol or attribute;
- Re-factoring, to optimize grammars generating smaller and more efficient versions.

Metrics are organized in three groups of assessment parameters:

- Size metrics, that measure the number of symbols, productions, and so on (grammar and parser sizes);
- Form metrics, that describe the recursion pattern and measure the dependencies between symbols (the grammar complexity);
- Lexicographic metrics, that qualify the clearness/readablity of grammar identifiers, based on a domain ontology.

Slicing operation builds partial grammar with the elements that derive in zero or more steps on the criterion (backward slicing), or that are reachable from the criterion (forward slicing). The criterion can be either a symbol or an attribute. Slices are usually presented as paths over the dependence graphs. Figures 8 (a) and (b) illustrate a forward and a backward slice w.r.t the symbol *age*.
Re-factoring is a not so usual functionality that transforms the original grammar into a minimal one, removing all the *useless productions*. Another transformation also provided is the generation of a new grammar combining forward and backward slices with respect to the same symbol (see Figure 8 (c)).

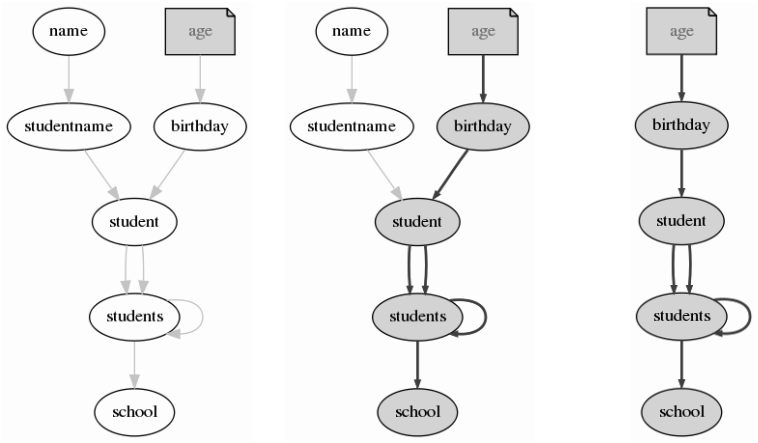


Fig. 8. Slices with respect to symbol *age*: (a) Forward slice; (b) Backward slice and (c) Combination of Forward and Backward slices

5 Conclusion

In the context of our Visual$\mathcal{LISA}$ project, we felt strongly impelled to use a universal meta-language for attribute grammars description in order to translate the graphical specification drawn in Visual$\mathcal{LISA}$, instead of producing a translation specific for a given compiler generator. As we did not find any notation with this tool-independence characteristic, we realized that a new one must be defined. As a mandatory requirement we stated that it should be used by our AG development environment, as well as it should be the *lingua-franca* of grammar-based tools. To satisfy the first part of the requirement it should allow to describe all elements present in a concrete AG that specifies the syntax of a language, and also those complementary definitions that permit the generation of a compiler for that language. To satisfy the second part of the above statement, the new meta-language should be supported on XML.

Conceived to annotate unstructured documents in a way independent of their future processing, XML immediately became the universal interchangeable data representation for assuring systems' interoperability. So our decision to design a specific XML-Schema for AGs, was obvious. Along the paper, we have shown how that design, was conducted systematically by the formal definition of attribute grammar.

Besides the presentation of $\mathcal{X}$AGra, the first goal of this paper, we also introduced $\mathcal{X}$AGraAl, a tool (completely independent of the one that motivates the conception of $\mathcal{X}$AGra) that supports grammar analysis taking as input a $\mathcal{X}$AGra grammar. The objective was to illustrate an applicability of this new XML dialect for AGs description.

Visual$\mathcal{LISA}$ environment is now generating $\mathcal{X}$AGra to translate a visual AG into a textual format usable by a compiler generator, or similar tool. The adaptation of the original $\mathcal{LISA}$ generator to the new one, that should now be called VisualAG, was an easy task performed very fast. However, for each generator that we want to couple to

VisualAG, it will be necessary to develop a $\mathcal{X}$AGra uploader, this is, a translator from $\mathcal{X}$AGra to the specific notation of the tool under consideration. As future work, the following translators are planed: $\mathcal{X}$AGra into LISA (a traditional LR parser generator); $\mathcal{X}$AGra into AntLR (an LL parser generator, based on an extended BNF grammar); $\mathcal{X}$AGra into Eli (an LR parser generator with special constructors). Also a translator from $\mathcal{X}$AGra to Yacc, could be a challenging project.
The creation of these translators is possible and easy. We are sure of this, because $\mathcal{X}$AGraAl showed its feasability. Moreover, $\mathcal{X}$AGraAl's front-end (the parser and semantic analyser that reads the $\mathcal{X}$AGra input and transforms it into an internal representation for further processing) would be similar to the core of these translators, so it can be reused.
To conclude, we claim that: (i) $\mathcal{X}$AGra is abstract and universal and (ii) this dialect was not crafted to be pleasant for human reading. Concerning the first point, we base that statement on the fact that $\mathcal{X}$AGra was derived from the formal and complete definition of Atrribute Grammar. Also, the dialect is abstract because it is completely independent from the concrete syntax of any compiler generator (CG). The second point is obvious: reading XML documents, although possible, is a cumbersome task for humans; the purpose of this dialect (generated by a tool) is to be processed by tools like $\mathcal{X}$AGraAl or the translators we plan for further work.

References

1. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems **2**(2) (June 1968) 127–145
2. Oliveira, N., Pereira, M.J.V., da Cruz, D., Henriques, P.R.: VisualLISA. Technical report, Universidade do Minho (February 2009) www.di.uminho.pt/~gepl/VisualLISA/documentation.php.
3. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An interactive environment for programming language development. Compiler Construction (2002) 1–4
4. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (August 2006)
5. GlassFish: Java architecture for XML binding. https://jaxb.dev.java.net/ (June 2009)
6. GlassFish: Java API for XML processing. https://jaxp.dev.java.net/ (June 2009)