# Lyntax – A grammar-Based Tool for Linguistics

**Manuel Gouveia Carneiro de Sousa**
University of Minho, Braga, Portugal

**Maria João Varanda Pereira**[1]
Research Centre in Digitalization and Intelligent Robotics,
Polythechnic Insitute of Bragança, Portugal

**Pedro Rangel Henriques**
University of Minho, Braga, Portugal

## Abstract

This paper is focused on using the formalism of attribute grammars to create a tool that allows Linguistic teachers to construct automatically their own processors totally adapted to each linguistic exercise. The system developed, named **Lyntax**, is a compiler for a domain specific language which intends to enable the teacher to specify different kinds of sentence structures, and then, ask the student to test his own sentences against those structures. The processor **Lyntax** validates the grammar (DSL program) written by the teacher, generating a processor every time the student defines a new sentence. For that ANTLR is used in both steps, generating not only the specialized processor but also the visualization of the syntax tree for analysis purposes. An interface that supports the specification of the language was built, also allowing the use of the processor and the generation of the specific grammar, abstracting the user of any calculations.

**2012 ACM Subject Classification** Human-centered computing → Visualization systems and tools; Social and professional topics → Computer science education; Software and its engineering → Source code generation

**Keywords and phrases** Attribute Grammars, Linguistic Rules, Pedagogical Linguistic Tools

**Digital Object Identifier** 10.4230/OASIcs.SLATE.2021.4

**Funding** This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the Projects Scopes: UIDB/05757/2020 and UIDB/00319/2020.

## 1 Introduction

Attribute Grammars are a way of specifying syntax and semantics to describe formal languages [3] and were first developed by the computer scientist Donald Knuth in order to formalize the semantics of a context-free language [7]. They were created and are still used for language developing, compiler generation, algorithm design and so on [8]. Due to its similarity with natural language grammars, it seems natural to explore the possibility to use them for linguistics purposes [4]. Using attribute grammars, it is possible to specify the way sentences are correctly written. For instance, using *synthesized attributes*, it is possible to represent the gender of an adjective, while *inherited attributes* can be used to associate the appropriate meaning to a preposition, depending on the context of the sentence [5]. In this way, linguistic rules can be modelled with an attribute grammar, and when a sentence is provided, it is possible to validate its syntax and contextual semantics, adverting for any errors that may be encountered [2].

Applying attribute grammars to model the syntax and semantics of natural languages is a technique that has already been practiced, but it demands knowledge in grammar engineering in order to translate natural languages properties to attribute grammar rules [3]. In spite of

[1] to mark corresponding author

the existing tools, they are not so easily available and straightforward for those who do not have programming and computation proficiency – in this specific case, linguists. There are tools available that use languages with logic components to define linguistic rules but they are not so easy to use. It is easier to rapidly grasp the concepts that are involved in this domain an create a domain specific language to allow the definition of linguistic rules in a higher level of abstraction.

So, the main proposal is to define a new DSL (Domain Specific Language) with a simpler notation, making it easy to understand and learn and to rapidly use. The main focus is to keep the syntax as simple and concise as possible, avoiding complex (or not so common) symbols. This allows the specification of linguistic rules to be done in a much natural manner. Also, a visually appealing user interface was created, granting the user the possibility of analysing the generated syntax-tree.

Moreover, since the teacher specification is agnostic it allows the student to map his sentence written in Portuguese or other natural language to the structure defined by the teacher. That's why the processor is regenerated for each student sentence.

Besides this introduction, this paper is divided in 5 sections: one focused on the state-of-the-art, another one to introduce **Lyntax**, a fourth one explaining two case studies and a last one before conclusion for implementation details.

## 2 State-of-the-art

As the project here reported aims at developing a tool which allows the definition of linguistic rules in a computational style, it is obvious that such a system can be used to improve the teaching of linguistics in a classroom context. In this section, it will be discussed available tools that can also be used for similar purposes, that deserve to be studied and referenced.

PAG is a tool that was created with the purpose of helping two distinct groups of students from *Universidad Complutense de Madrid*. One of those groups involves computer science students, attending a compiler construction course, and the other group involves linguistic students, from a class on computational linguistics. Teachers from both classes used the same methodology to teach their classes, and noticed that it wasn't good enough for the students to master all the concepts: On one hand, they would have computer science skilled students, with great aptitude to produce solutions, but leaving aside the respective specifications, which lead to poor and inaccurate formal specifications, but on the other hand, linguistic students produced good formal specifications, as they are proficient with the natural language, but lack computer science skills to well transpose all the knowledge into computacional models [6]. The result was an environment based in attribute grammars that allows the specification of those same grammars using a language close to Prolog. The main goal was to embed Prolog into the language and maintain all the familiar basic notation, since both groups of students were already familiar with the Prolog and attribute grammar syntax and notation. Through rapid prototyping, which PAG makes use of, it is possible to obtain a functional processor at a embryonic state of the problem [6]. With this, computer science students can obtain results quite early, allowing them to apply more time into formal specifications. Moreoever, as the complexity of the syntax is reduced, this allows for a better and easier learning experience for students which have less aptitude for the solution codification or programming in general, which is the case for linguistic students. Overall, PAG solved the problem that was purposed in an effective way. Despite that, and giving the respective credit to those who built the tool, the fact is that Prolog can still be quite difficult to grasp for some people, and a challenge when it comes to learn it. The usage of a specification language that closely resembles the natural one, could be a great addition.

CONSTRUCTOR [1] is a Natural Language Interface that accepts and processes English sentences, using them as instructions for plane geometry constructing. Those instructions are then transformed into the respective graphical representation. The idea is that these sentences are issued as commands which represent steps for the creation of a geometrical construction. CONSTRUCTOR analyses the issued input, translates it into a semantic representation and, based on that semantic representation, builds a visual construction. Furthermore, CONSTRUCTOR keeps track of the sequence of inputs issued by the user, which results in a more controlled construction process, while giving the user feedback within each step. The purpose of exploring a tool of this type is not directly related to linguistics nor linguistic rules training. The relevance of this reference is related to the use of attribute grammars with natural language processing, which techniques could be helpful when tacking a specific problem of this kind.

Overall, both tools, despite their differences, share the use of attribute grammars as a basis for a particular system, or even to simplify their use through various techniques. On account of that, it is compelling to create a friendly way to specify and analyse different linguistic rules and their examples, which intends to be the purpose of **Lyntax**.

## 3 Lyntax: a new proposal to specify linguistic rules

This section presents **Lyntax** as a DSL based solution to describe and test linguistic rules in natural language sentences.

In order to specify all kinds of possible sentences/rules, the idea of creating a "meta-language" emerged. This "meta-language" will be used by the teacher to specify the rules for sentence construction. These rules will be written (in a single file) according to the following structure, that is divided into three main components:

1. **STRUCTURE** – the block where the teacher specifies the structure of the sentences to be tested.
2. **ERRORS/RULES** – list of conditions that should be tested over the sentences. Using ERRORS, if the conditions are matched, an error will be thrown. On the other hand, using RULES, the conditions expressed need to be true to accept the sentence as a valid input.
3. **INPUT** – this block corresponds to the sentence (the lexical part) that the student wants to test. This will be written by the student and then automatically joined with the specification previously described by the teacher.

As can be seen in Figure 1, this file will then be processed by an ANTLR (ANother Tool for Language Recognition – a Compiler Generator) that will work on the information that was written, and then generate a grammar (also specified in ANTLR syntax). This grammar corresponds to the translation of the "meta-language" into ANTLR instructions. Afterwards, the generated grammar will be used to create a validator of sentences; the student can write his sentences (the INPUT that is used to generate the processor and the student's sentence represented in the Figure 1 are the same) and use that validator to obtain confirmation about the sentence correctness. A new processor will be generated for each sentence the student wants to test. The results would be the validation of the given sentences and a tree for a better visualization of the input structure.

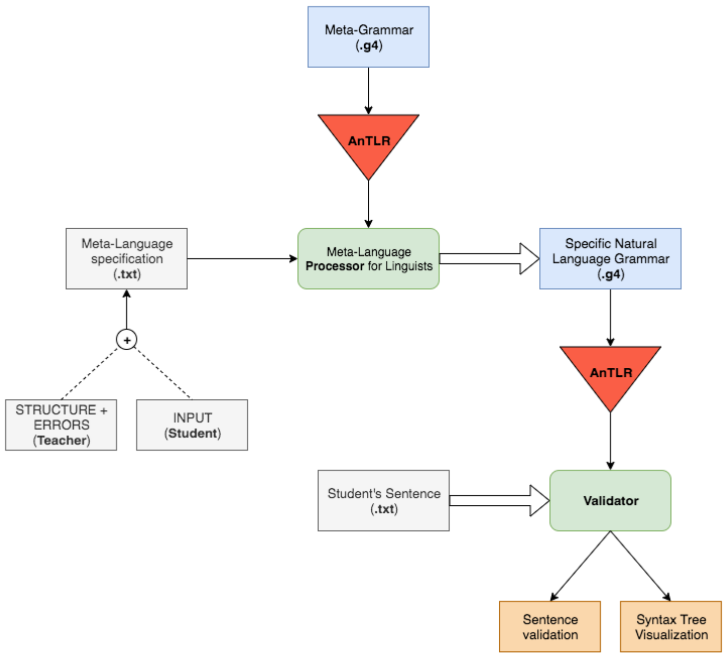

Meta-Grammar (.g4)
AnTLR
Meta-Language specification (.txt)
Meta-Language Processor for Linguists
Specific Natural Language Grammar (.g4)
+
STRUCTURE + ERRORS (Teacher)
INPUT (Student)
AnTLR
Student's Sentence (.txt)
Validator
Sentence validation
Syntax Tree Visualization

**Figure 1** System architecture.

## 3.1 Meta-Language

As it was stated in the introduction of this paper, the main project goal is to create a DSL that must be easy to learn and to grasp. With this in mind, the structure described is composed of three parts, where two of them will be written by the teacher, and the third one is intended to be written by the student; the three parts will be concatenated into a single file.

### 3.1.1 Domain Specific Meta-Grammar

The created DSL joins the teacher's specification with the student's sentence and its structure is described next.

**Listing 1** Processor production.

```
processor : structure errors input
;
```

Firstly, the teacher specification is divided into **structure** and **errors** blocks. The **structure** block is divided into the main **parts** of the sentence. Each part has an **element** containing the information about a certain component.

**Listing 2** DSL structure/part/element productions.

```
structure : 'STRUCTURE: ' (part)+ ;

part : 'part' '[' element ']' ;

element : '(' WORD ( '|' WORD )* ( ',' attributes )?
    ( ',' subparts )? ')' ('?')? ;
```

As seen in Listing 2, the **element** is composed of the name of the component, a possible set of **attributes** and possible **subparts**.

**Listing 3** DSL attributes/subparts productions.

```
attributes : 'attributes' '{' WORD ( ',' WORD )* '}'
;

subparts : 'subparts' '[' element ( ',' element ) ']'
;
```

The **subparts** production allows *injecting* more elements inside a single component. So, one component may be composed of several other components. As shown in Listing 3, the **subparts** production is a list of one or more elements.

Secondly, the teacher can define a list of restrictions to be applied to each attribute defined in the previous structure. A sentence will be valid if it follows the specified structure and if it obeys to the specified conditions.

**Listing 4** DSL errors/conditions productions.

```
errors : ('RULES'|'ERRORS') ':' ( condition ';' )+
;

condition : assignment ( ('AND'|'OR') assignment )*
;
```

The **errors** production will have two meanings“RULES’, then the conditions defined by the teacher need to be checked in order for a sentence to be correct; on the other hand, if the keyword is “ERRORS”, then if the conditions are matched, the sentence is not considered correct within that structure. As shown in Listing 5, the **condition** production is composed of a set of assignments that can be joined using the logical operators “AND” and “OR”. Each condition intends to create logical evaluations for the various attributes defined. Each assignment is composed of expressions.

**Listing 5** DSL assignment/expression production.

```
assignment
    : expression ('='|'!=') expression
    | expression ('='!'!=') '"' WORD '"'
;

expression : WORD ( '.' WORD )* '->' WORD
;
```

Each **expression** is composed of the path to a certain attribute, to a value or to other expression. If, for instance, the teacher says that an attribute is equal to some value, then the student can not use other value to that attribute – this would result in an error.

Finally, the **input** block (Listing 6), which corresponds to the students specification. This was treated as a different and separate DSL, as its main purpose was to identify the lexical parts of the sentence written by the student, allowing for a correct and non-subjective parsing of each word in the sentence.

**Listing 6** DSL input production.

```
input : 'INPUT:' phrase
;

phrase : ( '-' parts )+
;
```

Each phrase is composed of one or more **parts**, each of them holding various **blocks** (Listing 7), where all the information is stored. Inside, the name of the components and their required attributes must be specified. It is also important to notice that a correct path must be specified by the student. If the student specifies a component that is not declared in the structure defined previously by the teacher, then an error should be thrown.

**Listing 7** DSL parts/block/content productions.

```
parts : '(' block ( ',' block )* ')'
;

block : WORD content
;

content : (slice)? (attrs)? (parts)?
;
```

The student can specify the **slice** of the sentence that corresponds to the component that is being declared, and a set of associated attributes (**attrs**). Furthermore, it is possible to continue to define more **parts** within one part, just like the teacher's DSL **subparts**.

**Listing 8** DSL slice/attrs/evaluations/eval productions.

```
slice : ':' '``' (WORD)+ '"'
;

attrs : '[' evaluations ']'
;

evaluations : eval ( ',' eval )*
;

eval : WORD '=' '``' WORD '"'
;
```

Inside the **slice** production shown in Listing 8, a list of words can be written. These are the words that will then be used to build the lexical part of the generated grammar. Also, when specifying attributes, the student must assign a value for each attribute that will then be used to validate each component of the sentence.

## 4 Case Studies

In order to better explain the architecture proposed in the previous section, some concrete examples will be presented in this section. The main idea is to show the specifications used by the teacher and by the student and how the generated processor verifies the correcteness of the student sentences.

## 4.1 Case Study 1

The example showed in Listing 9 contains a structure that is composed of two main parts: a subject (Sujeito) and a predicate (Predicado). The subject is then subdivided into a possible determiner (Determinante) and a noun (Nome), which are then matched with a word (the lexical part identified by the student). The predicate is composed of a verb and a complement that is directly related to the verb. This complement (Complemento_Direto) is then composed of a possible determiner (Determinante) and a mandatory noun (Nome).

**Listing 9** Example of a possible sentence structure defined by the teacher.

```
STRUCTURE:
    part[(
        Sujeito,
        attributes{tipo},
        subparts[
            (Determinante)?,
            (Nome)
        ]
    )]

    part[(
        Predicado,
        subparts[
            (Verbo, attributes{tipo}),
            (Complemento_Direto, subparts[(Determinante)?, (Nome)]),
        ]
    )]

ERRORS:
    Sujeito->tipo = "animado"
        AND Predicado.Verbo->tipo = "inanimado";
    Sujeito->tipo = "inanimado"
        AND Predicado.Verbo->tipo = "animado";
```

In this particular example, both the subject and the verb from the predicate have an attribute named `tipo` which purpose is to check if the component is animated or inanimated. By analysing the ERRORS block (Listing 9), it can be seen that the value of the attribute `tipo` must be the same for both components, otherwise an error should be pointed out.

In this case, the sentence:

"O Carlos teme a sinceridade."

which is in fact a valid sentence, as the name "Carlos" and the verb "teme" are both of the type animated.

**Listing 10** Example of a correct student's sentence.

```
INPUT:
    - (Sujeito: "O␣Carlos" [tipo = "animado"]
        (Determinante: "O", Nome: "Carlos"))
    - (Predicado: "teme␣a␣sinceridade"
        (Verbo: "teme" [tipo = "animado"],
         Complemento_Direto: "a␣sinceridade"
            (Determinante: "a", Nome: "sinceridade")))
```

An example of an incorrect input sentence can be seen in Listing 11.

**Listing 11** Example of an incorrect student's sentence.

```
INPUT:
    - (Sujeito: "O␣Carlos" [tipo = "animado"]
        (Determinante: "O"))
    - (Predicado: "teme␣a␣sinceridade"
        (Verbo: "teme" [tipo = "animado"],
         Complemento_Direto: "a␣sinceridade"
            (Determinante: "a", Nome: "sinceridade")))
```

In this structure the `Nome` component is missing, an error message identifying the missing component will be printed to the output, as shown in Listing 12.

**Listing 12** Example error message of missing component.

```
ERROR: (INPUT)
- The mandatory component 'Nome' has not been defined.
```

## 4.2 Case Study 2

If, for instance, the main goal of the teacher is to test different attributes despite of the components of a sentence, a simple structure can be defined for that purpose. The following structure and rules aim at testing the gender conformance between two components, and this can be done with very simple sentences.

**Listing 13** Example of an arbitrary sentence structure.

```
STRUCTURE:
    part(
        Frase,
        subparts[
            (Determinante, attributes{genero}),
            (Nome, attributes{genero}),
            (Verbo)
        ]
    )

ERRORS:
    Frase.Determinante->genero = "masculino"
    AND
    Frase.Nome->genero = "feminino";

    Frase.Determinante->genero = "feminino"
    AND
    Frase.Nome->genero = "masculino";

    Frase.Determinante->genero != "masculino"
    AND
    Frase.Determinante->genero != "feminino";

    Frase.Nome->genero != "masculino"
    AND
    Frase.Nome->genero != "feminino";
```

Based on the rules written, we can see that the gender must be equal, or the sentence is invalid. Furthermore, the rules ensure that the gender can only be male or female ("masculino" and "feminino" respectively) in order to be a valid sentence. In Listing 14 an example of a possible valid sentence is presented.

**Listing 14** Example of an arbitrary sentence input.

```
INPUT:
    - (Frase: "A␣Olinda␣come"
        (Determinante: "A" [genero = "feminino"],
         Nome: "Olinda" [genero = "feminino"],
         Verbo: "come"))
```

A possible mistake that could be done in this particular example is a missing the specification of the attribute attribute `genero` for one of the components as can be seen in Listing 15.

**Listing 15** Example of an incorrect arbitrary sentence input.

```
INPUT:
    - (Frase: "A␣Olinda␣come"
        (Determinante: "A" [genero = "feminino"],
         Nome: "Olinda",
         Verbo: "come"))
```

Using the input written above (Listing 15), the error message of Listing 16 is displayed to the user.

**Listing 16** Example error message of missing attributes.

```
ERROR: (INPUT)
- There are attributes related to the component 'Nome'
  that were not defined.
```

## 5 Lyntax: Development

This section will present the development process of the system. Firstly, *OpenJDK* (Open Java Development Kit) was the *Java* platform chosen for that. Secondly, in order to process, execute or translate DSL programs, *ANTLR* **4.8** was used. Lastly, using the *Apache NetBeans*[2] (version **10**) IDE (Integrated Development Environment), it was possible to build the user interface that composes the system.

### 5.1 Meta-Grammar Processor

At first, the DSL grammar defined in 3.1.1 is used by *ANTLR* to generate a processor. This processor takes the teacher + student specification and constructs an *ANTLR* specification that will be used to generate a specific processor for each student exercise. This new processor is generated to be used by the student to verify if his sentences are correctly following the structure defined by the teacher. As consequence of that some errors like the ones presented in Listing 12 or Listing 16 may occur.

[2] `https://netbeans.apache.org/`

**Listing 17** Processor rule from the meta-grammar.

```
processor
@init {
    /* Main data structure. */
    List<RoseTree> struct = new ArrayList<>();

    (...)
}
    : structure[struct]
      errors[struct]
      input[struct]
    {
        (...)
    }
```

The structure presented in Listing 17 is responsible for storing all the information that is being parsed from the file given as input (the meta-language file).

The principle of having a tree as the main data structure falls into the need of maintaining a valid path. For example, if the teacher says that the structure will have a component ***A***, and this component has two children, ***B*** and ***C***, then the paths ***A***→***B*** and ***A***→***C*** should be stored. In this particular problem, it is required to have a tree that within each node has a list of children with an arbitrary size of ***N***. This type of structure is denominated as Rose Tree, which is a prevelant structure within the functional programming community. It is a multi-way tree, with an unbounded number of branches per node.

**Listing 18** RoseTree class.

```
class RoseTree {
    String chosenValue;
    String path;
    boolean visited;
    boolean required;
    Map<String, String> attributes;
    Set<String> optionValues;
    List<String> lexical_part;
    List<RoseTree> children;

    (...)
}
```

When in the main production (*processor*), a list of *Rose Trees* is initialized, with each tree of the list corresponding to the main components of the sentence. This structure would travel along the parsing tree, to first be populated with information and then serving as the main source of validation and checking.

On the first block (STRUCTURE) there are not many calculations happening within the productions. The main task is to simply validate the syntax and extract data to be stored in the *Rose Tree*. For each node, it is stored the name of the component, if it is required to be declared or not, a group of attributes (could be non-existent), a lexical part (if it is the case), and finally a list of nodes, referred as the children.

After the parsing of the structure, there is a list of conditions named ERRORS that need to be validated and converted into *Java* syntax – this conversion would then be injected on the main rule of the generated grammar. These logical expressions are based on the attributes of each component and their relations. For example, if the teacher says that a component ***A*** has an attribute named ***a***, and this attribute is required to have value ***x***, if the student assigns it a value of ***z***, then an error should appear. All these conditions can be combined with the logical operands "AND" or "OR". The way a logical expression is parsed is based on the path specified by the teacher when accessing the attribute. Using the example before, a component ***A*** with a child ***B***, with ***B*** having an attribute ***x***, in order to access it, the syntax should be

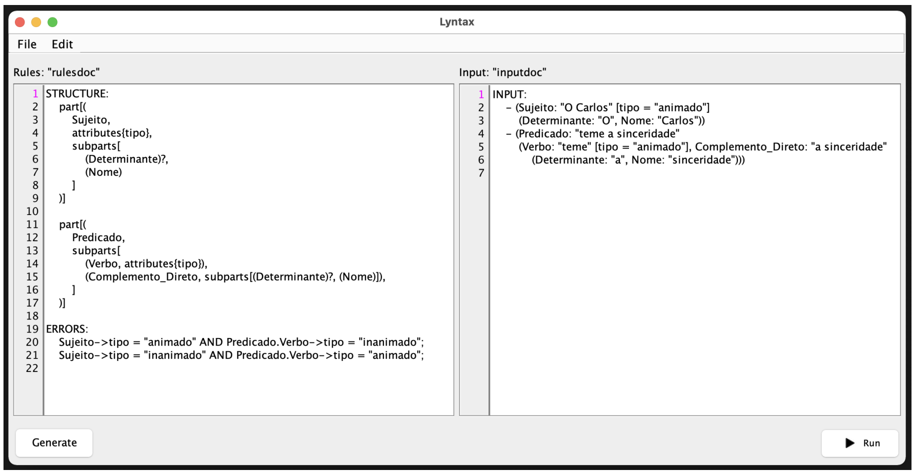
$$A.B \to x$$

as the full path is required. This is done in order to calculate the correct path and avoid ambiguity between attributes. Over the parsing of these rules, the path is being validated, and in case of any error, the user is notified.

Finally, the last block corresponds to the input that was written by the student. The goal is to validate the components that were defined, and match them with the structure created by the teacher. Again, the RoseTree was used as a way to check if the student's components and paths were valid. The task of the student was to "parse" his sentence and divide it by components, identifying the lexical segments and storing them within a node of the *Rose Tree*. At last, the main rule of the Meta-Grammar makes use of a generator to generate all the rules for the Specific Natural Language Grammar. Within this generator, the various *Rose Tree's* are passed as an argument and then traversed recursively.

## 5.2 Interface

As stated in the introduction of this paper, after the creation of a system capable of testing various sentences, the goal was to build a user interface that allowed for an easier and simpler use of the system, without the need of using the command line tool built, which takes care of runtime compilations. The interface was built using ***Swing***, a GUI widget toolkit for *Java*. *Swing* has a lot of sophisticated GUI components available for use, allowing the developer to focus on pure functionality (Figure 2). Furthermore, using the *Apache NetBeans* IDE for *Java*, it was possible to use a GUI builder for manipulating *Swing* components, by dragging and dropping them to a canvas – this would generate the specific *Java* code for each component.

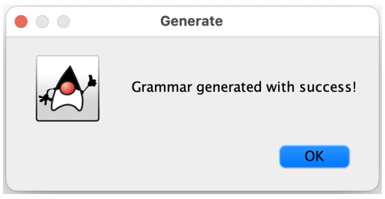


```
STRUCTURE:
   part[(
      Sujeito,
      attributes{tipo},
      subparts[
         (Determinante)?,
         (Nome)
      ]
   )]

   part[(
      Predicado,
      subparts[
         (Verbo, attributes{tipo}),
         (Complemento_Direto, subparts[(Determinante)?, (Nome)]),
      ]
   )]

ERRORS:
   Sujeito->tipo = "animado" AND Predicado.Verbo->tipo = "inanimado";
   Sujeito->tipo = "inanimado" AND Predicado.Verbo->tipo = "animado";
```

```
INPUT:
   - (Sujeito: "O Carlos" [tipo = "animado"]
      (Determinante: "O", Nome: "Carlos"))
   - (Predicado: "teme a sinceridade"
      (Verbo: "teme" [tipo = "animado"], Complemento_Direto: "a sinceridade"
         (Determinante: "a", Nome: "sinceridade")))
```

**Figure 2** Lyntax user interface.

After the specification of the rules (in the left side) and input (in the right side), the user can generate the Specific Natural Language Grammar to be able to create the Sentence Validator, using the "Generator" button. The text within the two text boxes is concatenated, and given as input for the MetaGrammar processor. All these operations are done in background, following the same order as the instructions showed above. If all goes well, the user should have prompted a message saying that the Grammar was successfully generated (Figure 3) – it is now possible to test the sentence.


Generate

Grammar generated with success!

**Figure 3** Grammar generation success message.

At last, by clicking the `Run` button, the validator is created, and the sentence passed as input. If no errors occur during this process, the user should see the sentence syntax tree as shown in Figure 4.

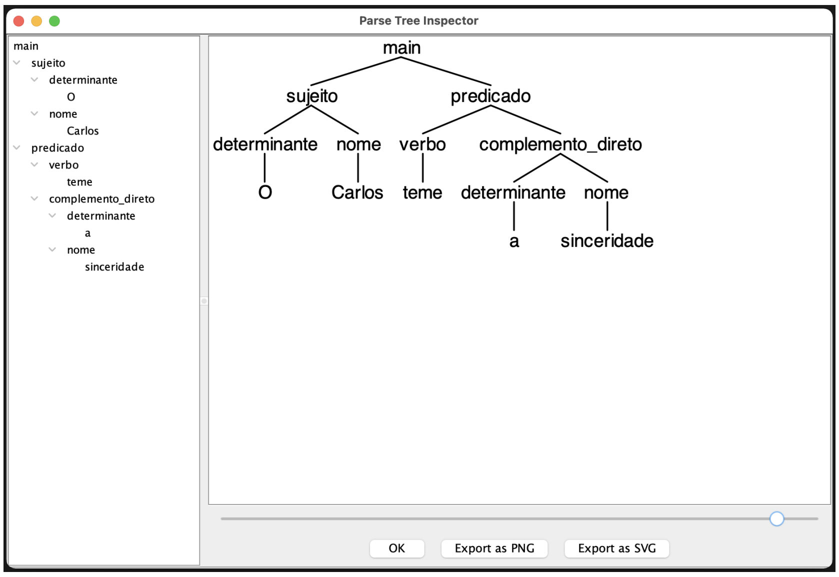


**Figure 4** Example of a generated syntax tree within TestRig.

## 6 Conclusion

The definition of a new well-rounded DSL was done in order to allow common users to define various kinds of sentence structures and linguistic rules. Using AG's, the processing and recognition of that DSL enabled for various types of computations to occur. Using ANTLR, we were able to generate a parser for this DSL that would perform many validations and generate a specific grammar for the sentence that is intended to be tested. Once again, using ANTLR, it is possible to generate a parser from the newly obtained grammar, allowing for the verification of the sentence and the visualization of the respective syntax tree.

The tool designed to provide these functionalities was named **Lyntax**. **Lyntax** is the name of the tool that merges the Meta-Language processor and validator with a user interface that allows for the specification of the language, such as an editor. This interface grants the user the possibility of generating the specific grammar and its respective parser at a high-level, with just the click of a button. The abstraction offered by **Lyntax** allows the user to focus only on the definition and testing of linguistic rules, which is, actually, his main concern.

One very important task that is still to be done is the conduction of tests with the final users – this could be both students from secondary schools or university. The main objective would be to see how the students would react and embrace the tool and its functionalities. For that, the users would be given a set of exercises to be executed using **Lyntax** and a survey with various questions that would try to capture their experience, but also query them about the usefulness of the tool and how it could help or enhance their linguistics studies within the classroom.

Moreover, the tool will be tested for both Portuguese and Spanish exercises in order to prove the versatility of the implemented approach and find commonalities between the defined rules.

## References

1 Zoltán Alexin. Constructor : a natural language interface based on attribute grammar. *Acta Cybernetica*, 9(3):247–255, January 1990. URL: `https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3371`.

2 Patrícia Amorim Barros, Maria João Varanda Pereira, and Pedro Rangel Henriques. Applying attribute grammars to teach linguistic rules. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

3 Rahmatullah Hafiz. *Executable attribute grammars for modular and efficient natural language processing*. PhD thesis, University of Windsor, Canada, 2011.

4 Petra Horáková and Juan Pedro Cabanilles Gomar. La concordancia nominal de género en las oraciones atributivas del español: una descripción formal con gramáticas de atributos. *Entrepalavras*, 4(1):118–136, 2014.

5 Donald E. Knuth. The genesis of attribute grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, pages 1–12, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

6 José Luis Sierra and Alfredo Fernández-Valmayor. A prolog framework for the rapid prototyping of language processors with attribute grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):19–36, 2006.

7 Kenneth Slonneger and Barry L Kurtz. *Formal syntax and semantics of programming languages*, volume 340. Addison-Wesley Reading, 1995.

8 Krishnaprasad Thirunarayan. Attribute grammars and their applications. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.