Universidade do Minho

Escola de Engenharia

Applying Attribute Grammars to teach Linguistic Rules

Manuel Gouveia Carneiro de Sousa

Supervisor:
Ph.D Pedro M. Rangel S. Henriques
Co-supervisor:
Ph.D Maria João T. Varanda Pereira

February 14, 2021

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Manuel Sousa

Acknowledgements

...

Abstract

This document presents a proposal for a Master Thesis within the topic of "Applying Attribute Grammars to teach Linguistic Rules", and will be accomplished at Universidade do Minho in Braga, Portugal.
This thesis is focused on using the formalisms of attribute grammars in order to create a tool to help linguistic students learn the different rules of a natural language. The main goal is to create a new DSL with a simple notation, suitable for any person that does not have any experience with common programming languages elements, that will allow to define linguistic exercices. Furthermore, it is expected to create a user interface that supports that same tool, granting a more visual experience for the user.

Keywords: Linguistic, Natural Language, Attribute Grammars

Resumo

Este documento refere-se a uma dissertação sobre o tópico “Aplicar Gramáticas de Atributos no ensino de Regras de Linguística”, e será concluída na Universidade do Minho em Braga, Portugal.
Esta dissertação pretende focar-se no uso dos formalismos das gramáticas de atributos de maneira a criar uma ferramenta que ajude os alunos de linguística a aprender as diversas regras da língua natural. O principal objetivo é a criação de uma DSL com uma notação simples, adequada a qualquer pessoa que não tenha experiência com elementos comuns das linguagens de programação, que permita a definição de exercícios de linguística. Para além disso, é esperada a criação de uma interface que sirva de suporte a essa mesma ferramenta, permitindo assim uma experiência visual ao utilizador.

Palavras-chave: Linguística, Língua Natural, Gramáticas de Atributos

Contents

List of Figures

List of Tables

List of Acronyms

DSL	Domain Specific Language
ANTLR	Another Tool for Language Recognition
CFG	Context-free Grammar
AG	Attribute Grammar

Chapter 1

Introduction

1.1 Context

Attribute Grammars are a way of specifying syntax and semantics to describe formal languages [7] and were first developed by the computer scientist Donald Knuth in order to formalize the semantics of a context-free language [16]. They were created and are still used for language developing, compiler generation, algorithm design, etc [17]. One other application would be the teaching of linguistic rules through the usage of formalisms presented in attribute grammars [9]. Using attribute grammars, it is possible to specify the way sentences are correctly written. By making use of "synthesized attributes", it would be possible to represent the gender of an adjective, while "inherited attributes" would be translated into the meaning of a preposition, depending on the context of the sentence [10]. There are an array of linguistic rules to be represented within an attribute grammar, and when a sentence is provided, it is possible to validate the syntax, adverting for any errors that may be encountered [3].

Applying attribute grammars to model all the different syntax and semantic behaviour of natural languages is a technique that has already been practiced, but it demands knowledge in programming syntax in order to translate natural languages rules to attribute grammar rules [7]. In spite of 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 which closely resembles logic, and use logic components, but it is easier to rapidly grasp the concepts of a domain specific language, that only does a list of tasks, than to use a language that is not created with a main purpose.

So, the main proposal is to define a new DSL (Domain Specific Language) with a much simpler notation, making it easy to learn and to rapidly understand. 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 rules to be done in a much natural manner. Also, it is desired to create a visually appealing user interface, granting the user the possibility of analysing the generated syntax-tree.

1.2 Objective

The main objective of this master thesis is to produce a pedagogical tool to support teaching linguistics. The detailed objetives are the following:

- Definition of simple and concise DSL, suitable for common users, to specify linguistic rules based in an attribute grammar.
- Construction of a tool that will have a friendly user interface and converts the DSL programs into attribute grammars using ANTLR (ANother Tool for Language Recognition) [1]. ANTLR will also be used in a second phase in order to take the generated attribute grammars and create the natural language sentence processor that will be used by the student.

[1] https://www.antlr.org/

1.3 Methodology

The research work will be performed at different stages. The methodology that will be followed to achieve this master project will focus on literature revision, solution proposal, implementation and testing. The following steps realise this methodology:

- Do a comprehensive research about attribute grammars in linguistics: what has been done, how has it been done, and ways to improve the previous work.
- Design a DSL that allows a straightforward specification of an array of rules.
- Develop a language translator that can translate programs written in the new language to ANTLR using ANTLR.
- Create a processor (using the generated attribute grammars of the last step) that will allow to receive sentences and check its correctness.
- Create an user interface that allows for a visual analysis of the generated syntax-tree.
- Experiment with some case studies, and test the tool with real linguistic students.

1.4 Document Structure

The document starts by introducing the problem, and, in **Chapter 1**, a context, objectives and methodology are presented. On **Chapter 2** the main concepts are briefly explained, followed by the presentation and explanation of current existing solutions that may help solving the current problem. **Chapter 3** consists on explaining the proposal for solving the stated problem, documenting the system architecture. **Chapter 4** is the process documentation of the development of the system, the paths taken to reach several solutions. **Chapter 5** displays a set of case studies and use cases for the tool, to demonstrate and validate its correct functionality **Chapter 6** intends to close the document with the overview of the work that was done, opinions on what was done, as well as suggestions on future work as a way of enhancing the solutions produced and presented in this documented.

Chapter 2

State of the Art

In this chapter is of the utmost importance to define some concepts that were used as a basis for this thesis. In addition, it will be discussed possible tools that already answer some of the questions that this thesis is trying to answer, and for that matter are important to be studied and referenced.

2.1 Context-free Grammar

A Grammar can be defined by the study and learning of the way sentences of a language are assembled. The word “grammar” is extracted from the Greek *Tékhnē grammatikē*), which has the meaning of “art of letters” [5]. This can be interpreted has the art of combining letters in order to produce a system which determines the correct way of constructing phrases. A grammar does not assign any meaning for these phrases (semantic), it only structures their form.

A Context-free Grammar (CFG) is defined by a tuple of 4 elements [8]

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

where **T** is the set of terminal symbols of a language; **N** is the set of non-terminal symbols; **S** is the start symbol of the grammar; **P** is the set of productions that compose this grammar. The productions within the grammar are rules of form

$$N \rightarrow (N \cup T)^*$$

where the left side, composed of a non-terminal symbol, may derive in a set of non-terminal and terminal symbols.

2.2 Attribute Grammars

Attribute Grammars were proposed by Donald Knuth in order to specify static and dynamic semantics of a programming language in a syntax-directed manner [17]. The process consists in constructing the syntax tree and then computing the values of attributes by visiting every single node. For each attribute it is possible to associate a domain of values, such as integers, strings or even complex structures.

Formally an attribute grammar (AG) is a tuple [14]

$$AG = (CFG, A, CR, CC, TR)$$

composed by a context-free grammar (CFG), which has been extended to provide context by using a set of attributes; The set of attributes (A), which exist in each production of a grammar, are divided into two groups, *Synthezized Attributes*, which allow values to be passed from one node to its parent, and *Inherited Attributes*, which allow values to be passed from the current node to a child [16]; rules for calculating attributes (CR) in all productions of the grammar; a set of contextual conditions (CC); and the transformation rules (TR) in all productions of the grammar.

2.3 Domain Specific Languages

A domain specific language is an "executable specification language, through appropriate notations and abstractions", usually restricted to a particular problem domain. Its objective is to improve productivity, and to allow solutions to be expressed in a more intuitive way and at the level of abstraction of the problem domain [18]. These types of languages provide a natural vocabulary for concepts that are fundamental to the problem scope [6], something that may lack when using a general-purpose language. DSLs are usually small and declarative, with very specific goals [18]. Refered as "little languages" [4], they are intended to solve problems within a specific domain, and not outside it.

One of the disadvantages of building a new DSL is the cost of their development, as it requires both domain and language development expertise [11], so the commitment of using DSLs as a solution for a software problem if often postponed or not even comtemplated. Nevertheless, as DSLs trade generality, they gain expressiveness (in a limited domain) - this results in the ease of repetitive tasks and smoother data description [12] and representation.

As discussed in [2], usability should be embedded in the DSL development process itself, and considered from the beginning of its development. The new created DSL needs to identify the problems within the domain, trying to overcome them while maintaining the expected expressivity.

2.4 PAG (Prototyping with Attribute Grammars)

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, involving computer science students, that attended a class which teached compiler contructions, and other group involving linguistic students, from a class on computacional 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 apititude to produce solutions, but leaving aside the respective specifications, which lead to poor and innacurate 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 [15].

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, the reason being 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 [15]. 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 the 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.

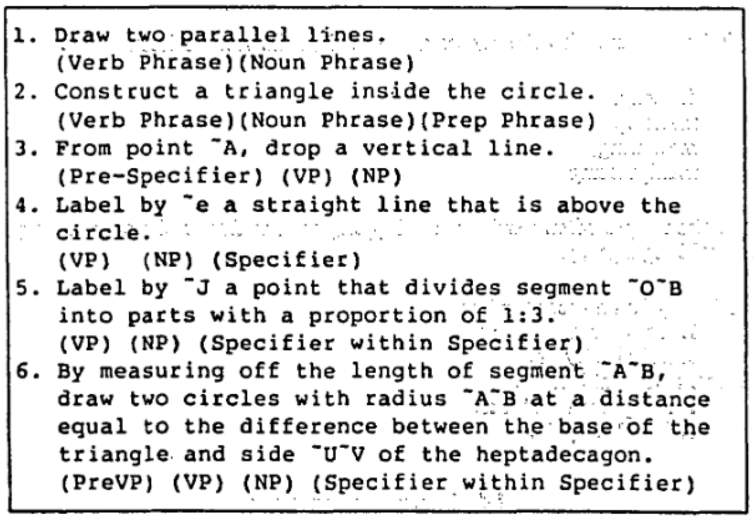
2.5 CONSTRUCTOR

CONSTRUCTOR [1] is a Natural Langugage 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.

CONSTRUCTOR is composed by the following parts:

- **Lexical Analyser** - consists in a machine dictionary of more than 300 items that are usually necessary for issuing commands. It also stores synonyms of the various items, as different people (based on age or level of training) may use different (but similar) words. This module is also extended by a morphological analyser, which analyses words that are not present in the dictionary, extracting its canonical lexeme an then perform an evaluation.

- **Syntactic Parser** - a string of tokens and terminals is used as input for the syntactic parser. This string is then processed into a sentence (or list of) with some structure assigned to it. All sentences must only describe one step and one step only, but nested sentences are possible. Below there is a table, extracted for the article ([1]) that shows various sentences examples.

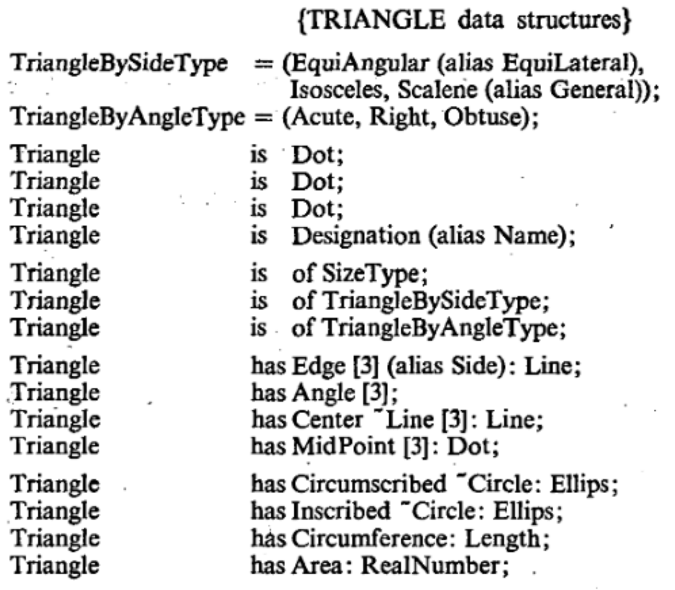


```
1. Draw two parallel lines.
   (Verb Phrase)(Noun Phrase)
2. Construct a triangle inside the circle.
   (Verb Phrase)(Noun Phrase)(Prep Phrase)
3. From point ~A, drop a vertical line.
   (Pre-Specifier) (VP) (NP)
4. Label by ~e a straight line that is above the
   circle.
   (VP)  (NP) (Specifier)
5. Label by ~J a point that divides segment ~O~B
   into parts with a proportion of 1:3.
   (VP) (NP) (Specifier within Specifier)
6. By measuring off the length of segment ~A~B,
   draw two circles with radius ~A~B at a distance
   equal to the difference between the base of the
   triangle and side ~U~V of the heptadecagon.
   (PreVP) (VP) (NP) (Specifier within Specifier)
```

Figure 2.1: Example of structured input sentences for CONSTRUCTOR.

- **Attribute Evaluation** - computes the basic features of grammatical structures, such as synthesized attributes. This computation will mostly involve synthesized attribute evaluation.

- **Semmantic Interpretation** - the semantic interpreter takes the results of all evaluations done before as input. The goal is to transform the English sentences into a *metalevel* description intended for building complex noun phrases.



```
                        {TRIANGLE data structures}
TriangleBySideType  = (EquiAngular (alias EquiLateral),
                       Isosceles, Scalene (alias General));
TriangleByAngleType = (Acute, Right, Obtuse);
Triangle              is  Dot;
Triangle              is  Dot;
Triangle              is  Dot;
Triangle              is  Designation (alias Name);
Triangle              is  of SizeType;
Triangle              is  of TriangleBySideType;
Triangle              is  of TriangleByAngleType;
Triangle              has Edge [3] (alias Side): Line;
Triangle              has Angle [3];
Triangle              has Center ~Line [3]: Line;
Triangle              has MidPoint [3]: Dot;
Triangle              has Circumscribed ~Circle: Ellips;
Triangle              has Inscribed ~Circle: Ellips;
Triangle              has Circumference: Length;
Triangle              has Area: RealNumber;
```

Figure 2.2: Example of a *metalevel* description of CONSTRUCTOR.

- **Construction Creator** - final component which takes a formal specification of an object, defining all the procedures to be executed. The results is a drawn geometric figure.

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.

2.6 VisualLISA (A Visual Programming Environment for Attribute Grammars)

VisualLISA is a visual programming environment created by Nuno Oliveira in the year of 2009 [13] for the specification of attribute grammars. Classified as a "Domain Specific Visual Language", its main goal was to enhance the front-end of one other tool named LISA [1], a compiler generator based in AG's that creates different visual tools based on the textual specification of the grammars.

The aim of this tool was to decrease the difficulty which is involved with specifying attribute grammars, not only for the LISA environment, but also regarding other types of systems, making specifications more visual and graphical.

Specifying grammars in a visual manner can be done using a set of icons (Figure 2.3) that must be combined to obtain the wanted result. Each icon or symbol has a unique function, and it is the users task to make the connections in a correct way.

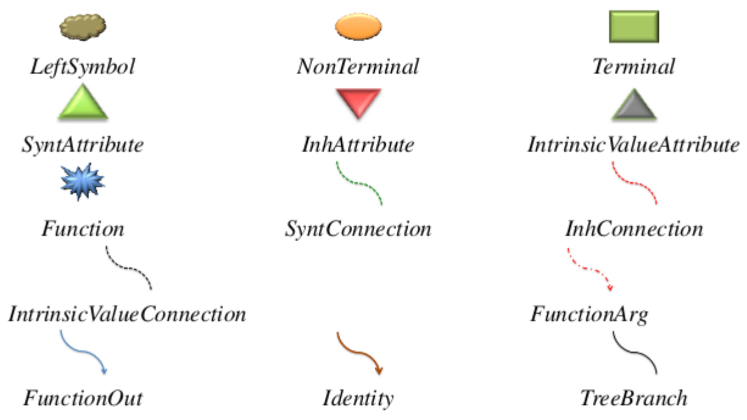


Figure 2.3: VisualLISA set of icons.

[1] https://labraj.feri.um.si/lisa/

The environment (Figure 2.4) consists in 4 windows, each one with an individual task: declare the productions of the grammar in a textual manner; declare functions, data-types, etc.; draw the grammar productions; specify computation rules that were previously declared.

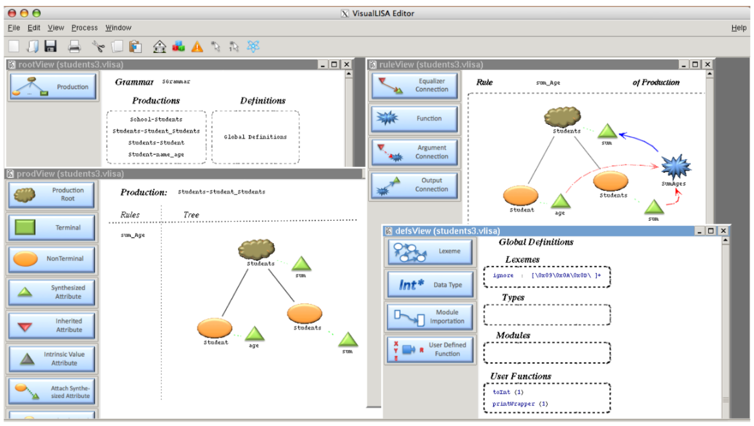


Figure 2.4: VisualLISA main window.

As an example, it was included a textual specification (Figure 2.5) and the respective graphical specification (Figure 2.6), extracted from the paper [13].

```
P1: Students → Student Students
        {Students0.sum = Student.age + Students1.sum}
P2: Students → Student
        {Students.sum = Student.age}
P3: Student → name age
        {Student.age = age.value}
```

Figure 2.5: Students textual grammar.

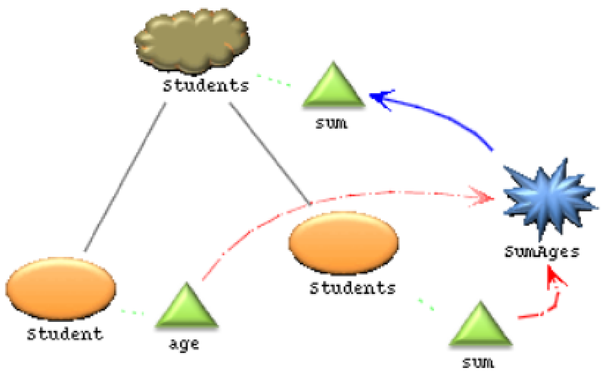


Figure 2.6: Students graphical grammar.

The main effect of the tool is not directly related to the problem that is trying to be solved, as the target users (linguists) may not be familiar with attribute grammars concepts such as terminal or non-terminal. This can cause some confusion when working with all the icons provided by the platform in order to create the intended atttribute grammar. Nevertheless, all the visual components that are associated could give some insights of interesting visual components to include/build when creating the proposed user interface to interact with the system.

2.7 Chapter Summary

In the previous sections, some tools were presented and major concepts discussed. Some of the tools may not have a direct impact on the proposal of this thesis, but their research and inclusion on this document were crucial. Each of the sections, despite their various differences, share the use of attribute grammars as a basis for a particular system, or even to simplify their use thorugh various techniques.

The solution proposed is an approach that, despite using attribute grammars, it is based on a new textual DSL using ANTLR and the automated generation of processors in order to validate sentences. In the next chapter, the proposal for this particular system will be discussed further.

Chapter 3

Lyntax: Proposal

The main purpose of this proposal is the definition of a DSL that allows the specification of all different kinds of sentences (done by the linguistic teacher), and afterwards, the possibility of the student to test his own sentences and check if they are written accordingly to the rules also previously specified by the linguistic teacher. One obstacle that was encountered was how to extract the lexical part of the sentence, and in what way would each component be classified. In fact, this task is quite subjective, as different components may have various definitions in one context. Having known this, the decision was made that the student would beforehand identify the lexical part of the sentence.

3.1 System Architecture

In order to specify all kinds of sentences/rules possible, 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 categories:

1. **STRUCTURE** - the block where the teacher will write how is the sentence supposed to be written, and what components will it have.
2. **ERRORS/RULES** - list of conditions that the teacher could write in order to be analysed afterwards, for example, certain values for different attributes. In the case of using ERRORS, if the conditions are matched, an error will appear. On the other hand, using RULES, the conditions need to be matched in order to have a valid input.
3. **INPUT** - this block corresponds to the "parsing" of the sentence (the lexical part) that the student wants to test. This will be written by the student and then automatically joined with the teachers information.

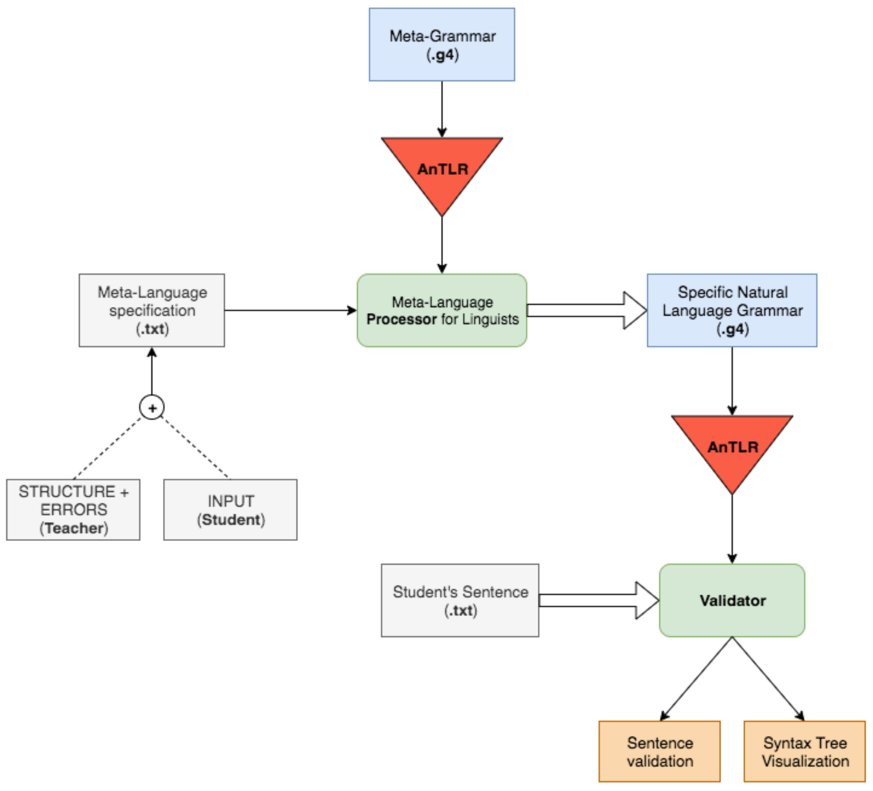

Meta-Grammar (.g4)
AnTLR
Meta-Language specification (.txt)
Meta-Language Processor for Linguists
Specific Natural Language Grammar (.g4)
+
STRUCTURE + ERRORS (Teacher)
INPUT (Student)
AnTLR
Student's Sentence (.txt)
Validator
Sentence validation
Syntax Tree Visualization

Figure 3.1: System architecture.

This file will then be processed by an ANTLR processor that will work on the information that was written, and then generate a grammar (also specified in ANTLR). 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, where the student can write his sentence/sentences and obtain results. A new processor will be generated for each sentence the student wants to test. The results would be the validation of the given sentence/sentences and a tree for a better visualization of the input structure.

3.2 Meta-Language

As it was stated in the beginning of this document, the main goal was to create a DSL that should be easy to learn and to rapidly understand and grasp. With this in mind, the structure mencioned before on the first section of this chapter was followed: Three main parts, where two of them would be constructed by the teacher, and the third one was intended to be written by the student and later concatenated in a single file.

3.2.1 Domain Specific Meta-Grammar

The main intention of this language is to preprocess the information written by the teacher + student and then generate a validator for a particular structure. With simplicity in mind, a first version of the DSL was created, and it will be explained next.

Listing 3.1: Processor production

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

Firstly, the teacher specification will be discussed - meaning the **structure** and **errors** blocks. The **structure** block is divided into **parts**, or main parts. These main parts correspond to the main components of the sentence. Each of these parts have an **element** within, containing the information about a certain component.

Listing 3.2: DSL structure/part/element productions

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

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

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

The **element** is composed by the name of the component, a possible set of **attributes** and possible **subparts**.

Listing 3.3: DSL attributes/subparts productions

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

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

The **subparts** production intends to be the path for “injecting” more elements inside a single component. One component may be composed by several other components. As shown in the example above, 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 3.4: DSL errors/expression productions

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

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

The **errors** production will have two meanings: if the keyword use is '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. The **condition** production is composed by 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. Conditions are composed by assignments, which are composed by expressions.

Listing 3.5: DSL condition production

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

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

The **assignment** production assigns an **expression**, which is composed by 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.

Thirdly, and finally, the **input** block, which corresponds to the students section. This was treated has 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 3.6: DSL input production

```
input : 'INPUT:' phrase
;

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

The sketch starts within a section named **phrase**, which corresponds to one sentence in particular. This production is composed by one or more **parts**, each of them holding various **blocks**, 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 3.7: DSL parts/component/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 attributes (**attrs**) that composes said component. Furthermore, it is possible to continue to define more **parts** within one part, just like the teacher's DSL **subparts**.

Listing 3.8: DSL slice/attrs/evaluations/eval productions

```
slice : ':' '“' (WORD)+ '”'
;

attrs : '[' evaluations ']'
;

evaluations : eval ( ',' eval )*
;

eval : WORD '=' '“' WORD '”'
;
```

Inside the **slice** production, 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.

For a better understanding of the three main categories (structure, errors and input), bellow there is an example that is based on the first case study, and shows what the specification of the teacher should look like.

Listing 3.9: Example of a possible sentence structure

```
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 the case of the student, this is the specification that should be used and one of the many examples that fit into the defined structured.

Listing 3.10: Example of the students parsing

```
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”)))
```

3.3 Chapter Summary

In this chapter, it was discussed the different phases of the proposed system architecture and what the output of a system like this should be. The principle is to create a new Domain Specific (Meta) Language that allows for the specification of sentence structures as well as proper input. Within this system, a processor for this Meta-Language is created by using ANTLR, which evaluates the specification written and is tasked with the generation of a specific natural language grammar. Combining the newly generated grammar with ANTLR, the result is a student's sentence validator that when given a sentence as input, performs all the necessary validations and, if correct, presents the user the respective syntax tree.

Lyntax, the word that combines the terms “Linguistics” and “Syntax”, was the name chosen for a system that combines both the meta-language processor and the user interface that makes use of such processor. The implementation of this system will be discussed in the next section.

Chapter 4

Lyntax: Development

This chapter will present the development and workflow of the system. As previously mentioned, the next step was to expand the defined DSL, and to use attributes as a form of calculation. Most of the productions were expanded, allowing for certain calculations to be injected over the tree.

It is important to identify the tools used to develop this system, as well as their respective versions. Firstly, *Java* was the language in which the system is based on. *OpenJDK* (Open Java Development Kit) is a free implementation of the *Java* platform, and a dependency for all the auxiliar tools. Within the development phase, version **11.0.9.1** of *OpenJDK* was used. Secondly, in order to process, execute or translate structured text (such as the DSL written), *ANTLR* was used for generating a parser from a previously written grammar. The system used version **4.8** of *ANTLR*. Lastly, using the *Apache NetBeans* [1] (version **10**) IDE (Integrated Development Environment), it was possible to build the user interface that composes the system.

[1] https://netbeans.apache.org/

4.1 Meta-Grammar

With the grammar divided into 3 main parts (STRUCTURE, ERRORS, INPUT), different types of calculations occur at different sections. The STRUCTURE and ERRORS blocks are written in a single file (by the teacher) which is then joined with the INPUT block (written by the student). The process starts with searching for the teacher and student specification, and then compiling the program using a meta-grammar based processor. A new processor is generated to be used by the student to verify if his sentences are correctly following the structure defined by the teacher. Within the grammar itself, the first rule,

Listing 4.1: Processor rule from the meta-grammar

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

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

starts by initializing the main data structure. This structure is responsible for storing all the information that is being parsed from the file given as input (the meta-language file).

When choosing the correct structure to store all the important data, the first approach taken was to store all components in a single Map, with each name of a component matching their respective value. The problem with this approach, which was identified right away, was that is possible to exist two or more components with the same exact name, causing a conflict within the Map. Furthermore, components have different information associated, like attributes, and it would be better if it is all in the same place - this created the need for a Component class.

The Component class would store the name of the component, a possible value and a Map that associated each attribute with some value. The components would all be stored within a List.

Listing 4.2: Component class

```
@members {
    class Component {
        String name;
        String lexical_part;
        Map<String, String> attributes;
    }

    /* Main data structure. */
    List<Component> struct;
}
```

The problem with this solution is that it does not follow any particular order (in this case, the STRUCTURE order), which can be very useful when validating the students. The sequence of components stored within a List would not be equal to the sequence of components that were defined in the structure previously defined.

The structure of the sentence takes a form of a tree, so that would be the correct way to store the information and maintain order. As each node could have less or more than two children, a binary tree was not the way to go. The idea was to build a Graph structure that used a mapped each node to a list of nodes.

Listing 4.3: Graph class

```
@members {
    class Graph
    {
        Map<Component, List<Component>> map;
    }
}
```

Although this could maintaing the order, the initial problem remainded. We could have components with the same exact properties, and this would cause conflict between edges, and not create a new node when it was supposed to.

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***.

Some backtracking was made to come up with an ideal solution. The prerequisites were that order needed to be maintained and each node (component) had an arbitrary number of ***N*** children. The previously created Component class would store all its values and a list of new components (children), creating a path between the parent component and said children. 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. This way, all the prerequisites would be matched, and all the information correctly stored.

Listing 4.4: 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 are 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 $\boldsymbol{A}$ has an attribute named $\boldsymbol{a}$, and this attribute is required to have value $\boldsymbol{x}$, if the student assigns it a value of $\boldsymbol{z}$, then an error should appear. All these conditions can be combined with the logical operands "AND" or "OR". The way that is parsed is based on the path specified by the teacher when accessing the attribute. Using the example before, a component $\boldsymbol{A}$ with a child $\boldsymbol{B}$, with $\boldsymbol{B}$ having a attribute $\boldsymbol{x}$, in order to access it, the syntax should be

$$A.B - >x$$

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.

4.2 Meta-Language Processor

In order to simplify the usage of the Meta-Grammar, and as the grammar itself made use of auxiliar *Java* classes, all of that was combined into a *JAR* file. Having this type of package would allow for a more flexible integration with any component. The Meta-Language Processor, which was created by providing the Meta-Grammar file to ANTLR, could now be used with the *JAR* file, providing as input the Meta-Language Specification.

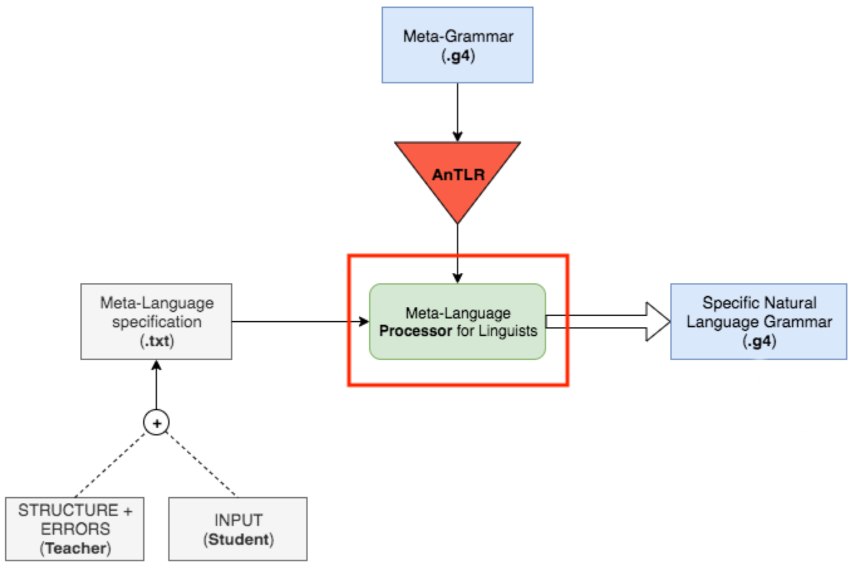

Meta-Grammar (.g4)
AnTLR
Meta-Language specification (.txt)
Meta-Language Processor for Linguists
Specific Natural Language Grammar (.g4)
+
STRUCTURE + ERRORS (Teacher)
INPUT (Student)

Figure 4.1: Excerpt of the system architecture - Meta Processor.

Using the command line, the instruction:

```
java -jar lib/MetaGrammar.jar input/meta-lang
```

tries to generate the Specific Natural Language Grammar, based on the input provided. In case of any error, the grammar would not be generated.

4.3 Specific Sentence Grammar Generator

As previously mencioned, the role of this generator is to produce the grammar that intends to recognize the students input. This grammar is specific to the sentence, and contains the conditions previously defined by the teacher ready to be evaluated. The generator is an auxiliar *Java* class, which contains the methods necessary to traverse the RoseTree given as argument, and create the strings of text for the grammar, which will then be appended to a file.

The principle of this generator is to create the independent strings first, and then recursively traverse the tree in order to create the productions themselves. This task was done using an auxiliar data structure,

```
Map<String, StringBuilder> productions
        = LinkedHashMap<>();
```

with each key representing the name of a production, and each value representing the various words that composed the rule of said production. The chosen structure would be a *Linked Hash Map*, as the insertion order was important to maintain. While traversing the tree, the information would be processed and also stored within the auxiliar Map. Lastly, all strings containing the productions, and also a lexer, are printed into a file of type ".g4". This is the file containing the grammar which will then be use to create the sentence validator.

4.4 Sentence Validator

If no errors occur in the previous step, we should now have a file named “Grammar.g4” that corresponds to the Specific Natural Language Grammar. This grammar contains all the tokens extracted from the Meta-Language specification, and combining it with ANTLR, we create a new specific Sentence Validator. When providing the student’s sentence to the Sentence Validator, and if all goes well, a Syntax Tree should be generated using a tool called *TestRig*. Using the command line once again, and providing a specific flag to the tool (*-gui*), we obtain the final syntax tree for the sentence provided:

```
java -cp \
        "lib/antlr-4.8-complete.jar:$CLASSPATH" \
        org.antlr.v4.gui.TestRig \
                Grammar main input/sentence \
        -gui
```

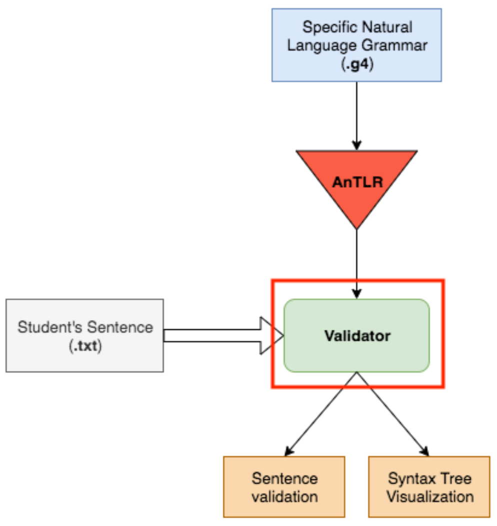

Specific Natural Language Grammar (.g4)
AnTLR
Student's Sentence (.txt)
Validator
Sentence validation
Syntax Tree Visualization

Figure 4.2: Excerpt of the system architecture - Sentence Validator.

As an example, using Listing 3.9 and Listing 3.10, the generated syntax tree would be:

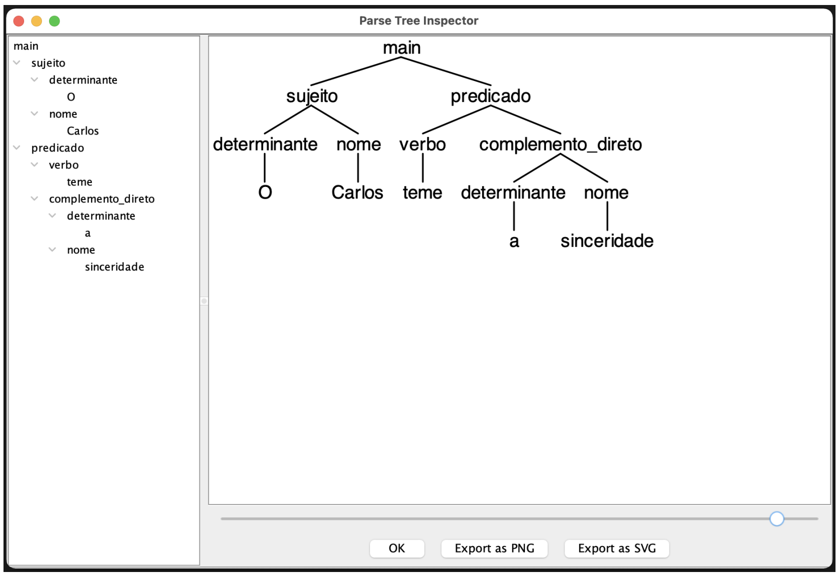
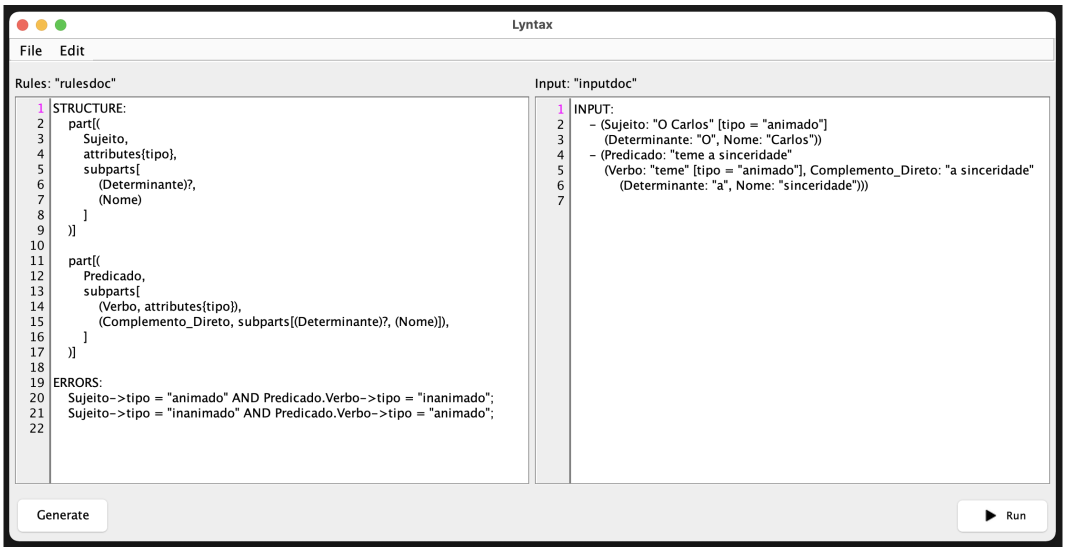


Figure 4.3: Example of a generated syntax tree within TestRig.

4.5 Lyntax: Interface

As stated in the introduction of this document, after the creation of a system capable of testing various sentences, the goal was to build an user interface that allowed for a more easy and simple use of said system, without the need of directly using the command line for providing inputs or manual 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. 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.

Objectively, the front-end part of the system would consist on a single window composed by two main text areas, corresponding to the rules and input blocks, one button to generate the specific sentence validator and one last button to inject the sentence into the validator and giving the user their sentence syntax tree. The window would also have a top menu bar that would allow the user of opening text files if desired. In any case, the user could write the STRUCTURE, ERRORS/RULES and INPUT blocks directly into the respective text areas without opening any file.



```
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")))
```

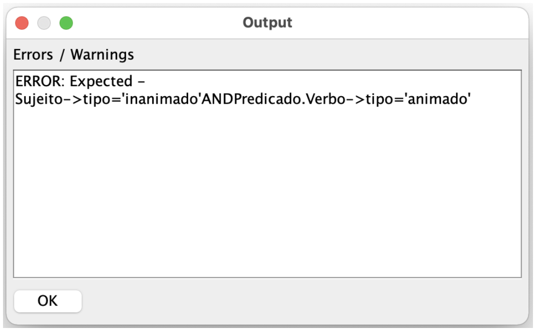
Figure 4.4: Lyntax user interface.

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 - it is now possible to test the sentence.

Grammar generated with success!

Figure 4.5: 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 the one used in Figure 4.3. On the other hand, if errors or warnings occur, they are displayed textually for the user in a small window.

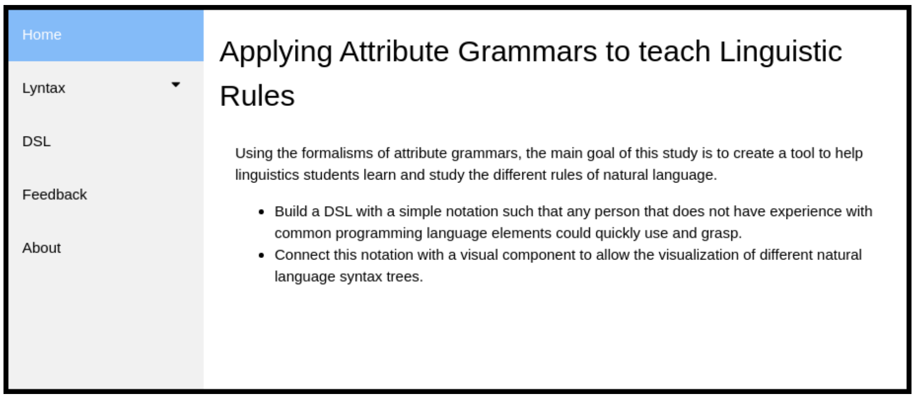


```
ERROR: Expected -
Sujeito->tipo='inanimado'ANDPredicado.Verbo->tipo='animado'
```

Figure 4.6: Example of an error message.

4.6 Lyntax: Website

In order to better spread some of the concepts discussed in this document, a website was created. This allows for the distribution of the system, and to also share the sources for the processor, the meta-grammar and other auxiliar files.

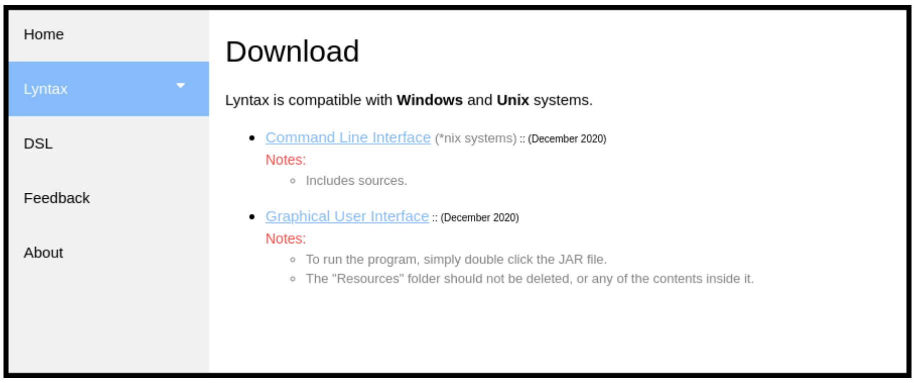


Applying Attribute Grammars to teach Linguistic Rules

Using the formalisms of attribute grammars, the main goal of this study is to create a tool to help linguistics students learn and study the different rules of natural language.

- Build a DSL with a simple notation such that any person that does not have experience with common programming language elements could quickly use and grasp.
- Connect this notation with a visual component to allow the visualization of different natural language syntax trees.

Figure 4.7: Lyntax Website Homepage.



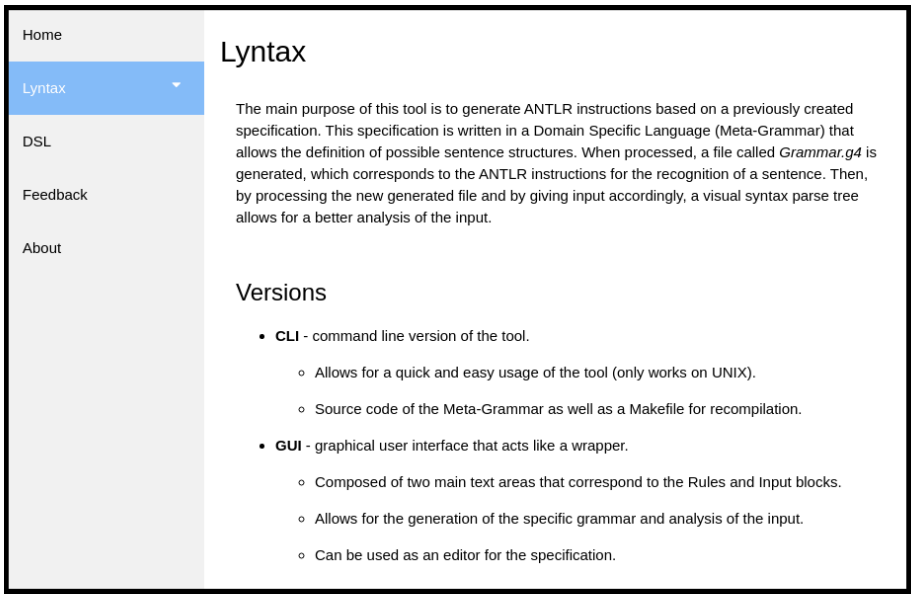
Download

Lyntax is compatible with **Windows** and **Unix** systems.

- Command Line Interface (*nix systems) :: (December 2020)
  Notes:
  - Includes sources.
- Graphical User Interface :: (December 2020)
  Notes:
  - To run the program, simply double click the JAR file.
  - The "Resources" folder should not be deleted, or any of the contents inside it.

Figure 4.8: Lyntax Website Download options.

The website includes a tab which explains the purpose of the tool and how it works in the background. The dependencies of the software are also mencioned.



Lyntax

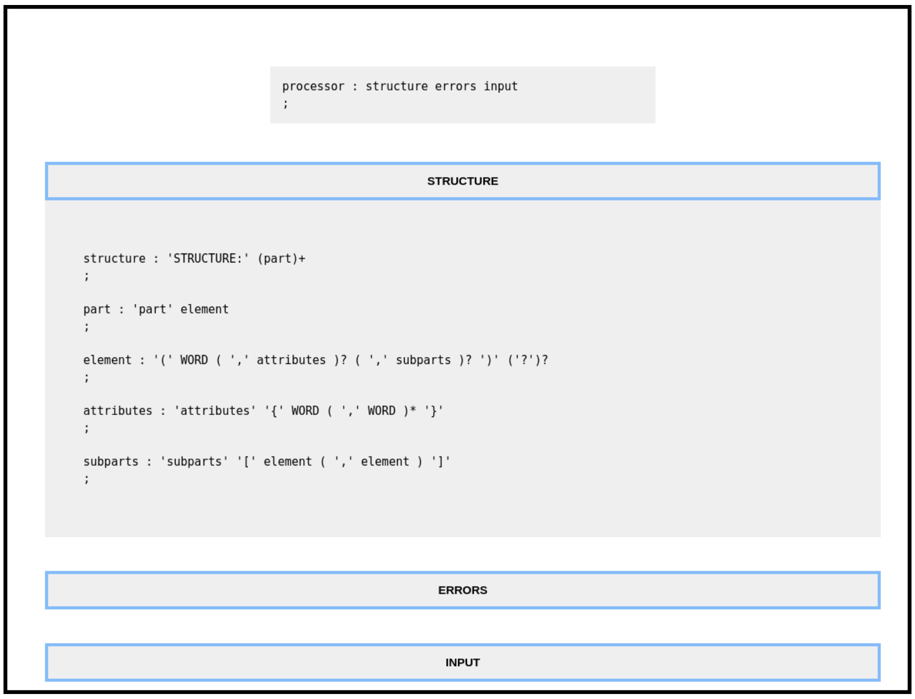
The main purpose of this tool is to generate ANTLR instructions based on a previously created specification. This specification is written in a Domain Specific Language (Meta-Grammar) that allows the definition of possible sentence structures. When processed, a file called *Grammar.g4* is generated, which corresponds to the ANTLR instructions for the recognition of a sentence. Then, by processing the new generated file and by giving input accordingly, a visual syntax parse tree allows for a better analysis of the input.

Versions

- **CLI** - command line version of the tool.
  - Allows for a quick and easy usage of the tool (only works on UNIX).
  - Source code of the Meta-Grammar as well as a Makefile for recompilation.
- **GUI** - graphical user interface that acts like a wrapper.
  - Composed of two main text areas that correspond to the Rules and Input blocks.
  - Allows for the generation of the specific grammar and analysis of the input.
  - Can be used as an editor for the specification.

Figure 4.9: Lyntax Website Information.

It was also included a tab for the visualization of the DSL created, for an easy access and analysis, and another tab for possible feedback of the tool.



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

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

part : 'part' element
;

element : '(' WORD ( ',' attributes )? ( ',' subparts )? ')' ('?')?
;

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

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

Figure 4.10: Lyntax Website DSL page.

4.7 Chapter Summary

At this stage, the development of the system is concluded. The result was, firstly, a DSL that allows for the specification of sentence structures and rules, as well as a respective input for those same structures. Secondly, the core system which processed the Meta-Language specification, generated a specific grammar, and then used that grammar to create a validator for the analysis and evaluation of a specified sentence. At last, and as a way of abstracting the complexity of the core system, the front-end intends to deliver a more flexible and straight forward use of the tool, allowing the user to focus on the description of the rules and testing of sentences.

The main challenges faced over the development period were, firstly, to choose the most appropriate data structure which would then be used to manipulate and process input data. As previously discussed, this structure demanded for a set of specified requirements to be met in order to produce correct results, so this task took quite a few tries and experiments to get right and accurate. Lastly, the error handling of the tool may cause some confusion at first, as it is handled and processed within the Meta-Grammar processor. This error message is then passed onto the specific natural language grammar, and printed to the user into a formatted window to ease the process. Nevertheless, the rules errors are precise and easy to understand.

Within the next chapter, some case studies will be explored in order to prove the correct functioning of the system.

Chapter 5

Case Studies

In order to better explain the solution architecture presented in this thesis, and also validate the work previous presented, 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 generator processor verifies the correcteness of the student sentences. The DSL design consists in slicing the sentence into parts, and each part can have subparts.

These case studies are also relevant as an example for a possible pedagogical scenario by using the system in a classroom context. The purpose of this chapter is to also demonstrate some of the process in which both the Teacher and Student will be involved, and to exhibit some common structures and inputs as well as their results.

5.1 Attribute Validation

This case study intends to demonstrate the validation on sentence components based on their attributes. The example showed in the previous chapter (Listing 3.9) contains a structure that is composed by 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 by a verb and a complement that is directly related to the verb. This complement (Complemento_Direto) is then composed by a possible determiner (Determinante) and a mandatory noun (Nome).

In this particular example, both the subject and the verb from the predicate have an attribute named 'tipo' which purpose is to check if each of the components are animated or inanimated. By analysing the logic in the ERRORS block (Listing 3.9), we can see that if the value of the attribute 'tipo' is different between the two components, than an error should be pointed. In this case, the sentence parsed by the student is

"O Carlos teme a sinceridade."

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

When running the example above (Listing 3.9 plus Listing 3.10) in the Meta Grammar processor, and if no errors occur, a specific grammar (in ANTLR) in then generated. For this specific case, this is the generated grammar.

Listing 5.1: Example of a specific generated grammar.

```
grammar Grammar;

@members {
    final String Sujeito__TIPO = “animado”;
    final String Predicado__Verbo__TIPO = “animado”;
}


main : sujeito predicado
{
    if ( Sujeito__TIPO.equals(“animado”) &&
         Predicado__Verbo__TIPO.equals(“inanimado”) )
       { System.out.println(“ERROR!”); }

    if ( Sujeito__TIPO.equals(“inanimado”) &&
         Predicado__Verbo__TIPO.equals(“animado”) )
       { System.out.println(“ERROR!”); }
}
;

sujeito : (determinante)? nome
;

determinante : 'O' | 'a'
;

nome : 'Carlos' | 'sinceridade'
;

predicado : verbo complemento_direto
;

complemento_direto : (determinante)? nome
;

verbo : 'teme'
;


/* LEXER */
(...)
```

Within the ‘main’ production, we can see the logical conditions that are ready to be evaluated when running this grammar. As the conditions are false, no errors should occur, allowing for the visualization of the syntax tree.

5.2 Missing components & Warnings

Still based on the previous defined structure (Listing 3.9), the student’s specification can still be missing some components that are mandatory.

This case study just intends to show the way that errors or warnings are notified to the user.

In order to demonstrate this, the input block defined above (Listing 3.10) is going to suffer some modifications in order to trigger errors or warnings.

Listing 5.2: Example of the students parsing with missing component

```
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 example, we can see that within the subject component (Sujeito), the noun component (Nome) is not defined. This particular case would cause an error as the noun component is mandatory (based on the previous structure Listing 3.9). The error message identifies the missing component.

Listing 5.3: Example error message of missing component

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

Another possible error would be to not define attributes, and to not give those attributes values. In this case, if we remove the attribute ‘tipo’ from the subject component (Sujeito), Listing 5.4: Example of the students parsing with missing attribute

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

the error should notify the user that the subject component is missing attributes, as attributes are always mandatory (if the component related to them is also mandatory).

Listing 5.5: Example error message of missing attributes

```
ERROR: (INPUT)
– There are attributes related to the component 'Sujeito'
   ↪ that were not defined.
```

When it comes to warnings, there is only one case that raises them. This happens when the user defines the same attribute multiple times, warning that only the last value will be considered for the final evaluation. If, for example, we use the same attribute twice on the subject component, Listing 5.6: Example of the students parsing with the same attribute in a single component

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

a warning is raised to notify the user that only the last value was considered as final (*tipo = “inanimado”*).

Listing 5.7: Example warning message of same attribute in a single component

```
WARNING: (INPUT)
- The attribute 'tipo' has already been declared! Using the
   ↪ last value found.
```

5.3 Arbitrary Structure

This last case study has the intention to demonstrate that is possible to define any arbitrary sentence structure, without obeying to any specific linguistic rules. 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 same purpose. The following structure and rules intend to test the gender between two components, and this can be done with very simple sentences.

Listing 5.8: 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.

Listing 5.9: Example of an arbitrary sentence input

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

Combining all the information in the processor, we generate a specific grammar for this arbitrary structure.

Listing 5.10: Example of a specific generated grammar.

```
grammar Grammar;

@members {
    final String Frase__Determinante__GENERO = "feminino";
    final String Frase__Nome__GENERO = "feminino";
}


main : frase
{
    if ( Frase__Determinante__GENERO.equals("masculino") &&
         Frase__Nome__GENERO.equals("feminino") )
       { System.out.println("ERROR!"); }

    if ( Frase__Determinante__GENERO.equals("feminino") &&
         Frase__Nome__GENERO.equals("masculino") )
       { System.out.println("ERROR!"); }

    if ( !Frase__Determinante__GENERO.equals("masculino") &&
         !Frase__Determinante__GENERO.equals("feminino") )
       { System.out.println("ERROR!"); }

    if ( !Frase__Nome__GENERO.equals("masculino") &&
         !Frase__Nome__GENERO.equals("feminino") )
       { System.out.println("ERROR!"); }
}
;

frase : determinante nome verbo
;

determinante : 'A'
;

nome : 'Olinda'
;

verbo : 'come'
;


/* LEXER */
(...)
```

As an example, we can also write the previous rules (Listing 5.8) using the keyword 'RULES' instead of 'ERRORS'. Using this keyword, the conditions would be interpreted in a different manner - all of them need to be true in order to considered the input as valid.

Listing 5.11: Example of an arbitrary sentence structure with rules

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

RULES:
    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”;
```

Changing the keyword would result in a slightly different specific generated grammar, because the conditions within the ‘main’ production need to be evaluated differently. When generating the grammar, if ‘RULES’ was the keyword chosen, we negate all conditions - if the condition is not true, it should cause an error, as the conditions, this time, need to be obeyed.

Listing 5.12: Different conditions from RULES block

```
(...)

main : frase
{
    if ( !( Frase__Determinante__GENERO.equals("masculino") &&
            Frase__Nome__GENERO.equals("feminino") ) )
       { System.out.println("ERROR!"); }

    if ( !( Frase__Determinante__GENERO.equals("feminino") &&
            Frase__Nome__GENERO.equals("masculino") ) )
       { System.out.println("ERROR!"); }

    if ( !( !Frase__Determinante__GENERO.equals("masculino") &&
            !Frase__Determinante__GENERO.equals("feminino") ) )
       { System.out.println("ERROR!"); }

    if ( !( !Frase__Nome__GENERO.equals("masculino") &&
            !Frase__Nome__GENERO.equals("feminino") ) )
       { System.out.println("ERROR!"); }
}
;

(...)
```

This example shows that the meta-language created is flexible to the point of writing arbitrary sentences or rules, augmenting the possibilities of syntactic structures.

5.4 Further examples and structures

In order to demonstrate even further the capabilities of the tool, some more examples of sentences with growing complexity will be included. Some of the previous examples had the intent of demonstrating validations and/or trigger errors and warnings. These next examples intend to explore structures with a bit more complexity.

This first example is, again, composed by a Subject - Predicate structure, with some caracteristics to the Predicate itself. Within this Predicate, we defined a Direct Complement (*Complemento_Direto*) and a Predicative to this Complement. This Predicative (*Predicativo_Complemento_Direto*) intends to give a specific caracteristic to the Complement itself.

Listing 5.13: Example of a sentence structure

```
STRUCTURE:
    part[(Sujeito)]
    part[(
        Predicado,
        subparts[
            (Verbo),
            (Complemento_Direto),
            (Predicativo_Complemento_Direto)?
        ]
    )]
```

Listing 5.14: Example of a sentence input

```
INPUT:
    - (Sujeito: “O rapaz”)

    - (Predicado (
        Verbo: “viu”,
        Complemento_Direto: “o homem”,
        Predicativo_Complemento_Direto: “com o telescopio”
    ))
```

The next example uses an attribute to limit the domain of a component (*Complemento_Circunstancial*). The attribute intends to enhance the meaning of the component by giving it a type (*tipo*). In this particular case, within the ‘ERRORS’ block, the attribute needs to have the value “lugar”, in order to represent a certain *place*, and to augment the context of the sentence.

Listing 5.15: Example of a sentence structure

```
STRUCTURE:
    part[(
        Frase, subparts[
            (Modificador),
            (Sujeito),
            (Predicado, subparts[
                (Verbo),
                (Complemento_Direto),
                (Complemento_Circunstancial, attributes{tipo})
            ])
        ]
    )]

ERRORS:
    Frase.Predicado.Complemento_Circunstancial->tipo != “lugar”;
```

Listing 5.16: Example of a sentence input

```
INPUT:
    - (Frase (
        Modificador: “Hoje”,
        Sujeito: “eu”,
        Predicado (
            Verbo: “comi”,
            Complemento_Direto: “uma pizza”,
            Complemento_Circunstancial:
                “na pizzaria abaixo” [tipo = “lugar”]
        )
    ))
```

We can also use an | (*or*) operator within the structure rules, giving the student the possibility of defining one or other component, but maintaing the main structure of the sentence. Using an operator with this capability, we prevent the need of creating two separate structures only for a single change.

Listing 5.17: Example of a sentence structure

```
STRUCTURE:
    part[(
        Frase, subparts[
            (Sujeito)?,
            (Predicado, subparts[
                (Verbo),
                (Complemento_Circunstancial_Modo |
                 Complemento_Circunstancial_Lugar)
            ])
        ]
    )]
```

Listing 5.18: Example of a sentence input

```
INPUT:
    - (Frase (
        Sujeito: “Eu”,
        Predicado (
            Verbo: “estive”,
            Complemento_Circunstancial_Lugar: “no Porto”
        )
    ))
```

Below is an example of a structure which supports more content within a sentence. In this particular case we are using two predicates (*Predicado*), both joined by a conjunction (*Conjuncao*). This example intends to demonstrate the capability of the tool when it comes to even larger structures. The principal remains the same, it all depends on the rules defined by the teacher.

Listing 5.19: Example of a more complex structure

```
STRUCTURE:
    part[(
        Frase, subparts[
            (Sujeito),
            (Predicado1, subparts[
                (Verbo),
                (Complemento_Indireto)
            ]),
            (Conjuncao),
            (Predicado2, subparts[
                (Verbo),
                (Modificador_Verbal)
            ])
        ]
    )]
```

Listing 5.20: Example of a more complex sentence input

```
INPUT:
    - (Frase (
        Sujeito: “Los soldados”,

        Predicado1 (
            Verbo: “dispararon”,
            Complemento_Indireto: “a los sentenciados”
        ),

        Conjuncao: “y”,

        Predicado2 (
            Verbo: “cayeron”,
            Modificador_Verbal: “muertos”
        )
    ))
```

As an example, it it also possible to have two identical structures, which both support the same kind of sentences, joined by a connector to create some kind of meaning as can be seen in Listing 5.21 and Listing 5.22.

Listing 5.21: Example of a another more complex structure

```
STRUCTURE:
    part[(
        Frase, subparts[
            (Sujeito1),
            (Predicado1, subparts[
                (Verbo),
                (Complemento_Circunstancial_Lugar)
            ]),
            (Conjuncao),
            (Sujeito2),
            (Predicado2, subparts[
                (Verbo),
                (Complemento_Circunstancial_Lugar)
            ])
        ]
    )]
```

Listing 5.22: Example of another more complex sentence input

```
INPUT:
    − (Frase (
        Sujeito1: "Juan",

        Predicado1 (
            Verbo: "vive",
            Complemento_Indireto: "en el edificio blanco"
        ),

        Conjuncao: "y",

        Sujeito2: "Maria",

        Predicado2 (
            Verbo: "trabaja",
            Modificador_Verbal: "aqui"
        )
    ))
```

5.5 Chapter Summary

The intent of this chapter was to validate the system developed and to demonstrate the capabilities of the tool. Through various examples and use cases, it was possible to exhibit how the tool handles certain input and how it responds to the user.

The most relevant aspect of the case studies are how they ilustrate the different scenarios in which they will be used. The target user are both the teacher, who is responsible for creating and implementing structures in which he/she would like to use for testing within a classroom context; the student, whose task is to create sentences that match with the rules defined, therefore practicing different linguistic aspects. The system as a whole can be considered functional for the use cases that intends to solve.

Chapter 6

Conclusion

...

6.1 Future Work

...

Bibliography

[1] Z. Alexin. Constructor : a natural language interface based on attribute grammar. *Acta Cybernetica*, 9(3):247–255, Jan. 1990.

[2] A. Barisic, V. Amaral, M. Goulão, and B. Barroca. How to reach a usable dsl? moving toward a systematic evaluation. *Electronic Communications of the EASST*, 50, 01 2012.

[3] P. A. Barros, M. J. Varanda Pereira, and P. R. 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.

[4] J. Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721, 1986.

[5] E. Britannica. Grammar. 02 2020. Accessed on 2020-12-21.

[6] D. Bruce. What makes a good domain-specific language? apostle, and its approach to parallel discrete event simulation. pages 17–35, 1997.

[7] R. Hafiz. *Executable attribute grammars for modular and efficient natural language processing.* PhD thesis, University of Windsor, Canada, 2011.

[8] P. R. Henriques. Brincando às linguagens com rigor: Engenharia gramatical. 2011. Habilitation in Computer Science (Technical Report), Dep. de Informática, Escola de Engenharia da Universidade do Minho,

habilitation monography presented and discussed in a public session held in April 2012 at UM/Braga.

[9] P. Horáková and J. P. C. 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.

[10] D. 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.

[11] T. Kosar, P. E. Martı, P. A. Barrientos, M. Mernik, et al. A preliminary study on various implementation approaches of domain-specific language. *Information and software technology*, 50(5):390–405, 2008.

[12] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–, 12 2005.

[13] N. Oliveira, M. J. V. Pereira, P. R. Henriques, D. da Cruz, and B. Cramer. Visuallisa: A visual environment to develop attribute grammars. 2009.

[14] M. J. V. Pereira, J. Fonseca, and P. R. Henriques. Ontological approach for dsl development. *Computer Languages, Systems & Structures*, 45:35–52, 2016.

[15] J. L. Sierra and A. 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.

[16] K. Slonneger and B. L. Kurtz. *Formal syntax and semantics of programming languages*, volume 340. Addison-Wesley Reading, 1995.

[17] K. Thirunarayan. Attribute grammars and their applications. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[18] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.