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

July 20, 2020

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

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. 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. 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 Hafiz (2011) and were first developed by the computer scientist Donald Knuth in order to formalize the semantics of a context-free language Slonneger and Kurtz (1995). They were created and are still used for language developing, compiler generation, algorithm design, etc Thirunarayan (1990). One other application would be the teaching of linguistic rules through the usage of formalisms presented in attribute grammars Horáková and Gomar (2014). 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 Knuth (1990). 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 Barros et al. (2017).

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 Hafiz (2011). 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 close as possible to a natural language, avoiding common programming languages elements (such as semicolons, curly brackets, etc.). 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 that pedagogical tool, with a friendly user interface, based on the reffered attribute grammar using ANTLR (ANother Tool for Language Recognition).

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.
- Research the principles of linguistic rules in different languages.
- 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.
- 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, giving some examples

and then using the tool produced. **Chapter 4** intends to close the document with summary of the work that was done, opinions on what was done, and explains what are the next steps to take in order to achieve the main objective.

Chapter 2

State of the art

In this chapter is of the utmost importance to define some concepts that were used has 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 Domain Specific Language

A domain specific language is an "executable specification language, through appropriate notations and abstractions", usually restricted to a particular problem domain. Their objective is to improve productivity, and to allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain van Deursen et al. (2000).

This types of languages provide a natural vocabulary for concepts that are fundamental to the problem scope Bruce (1997), something that may lack when using a general-purpose language. DSLs are usually small and declarative, with very specific goals van Deursen et al. (2000). Refered as "little languages" Bentley (1986), they are intended to solve problems within a specific domain, and not outside it.

2.2 Attribute Grammar

Attribute Grammars were proposed by Donald Knuth in order to specify static and dynamic semantics of a programming language in a syntax-directed manner Thirunarayan (1990). 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 Pereira et al. (2016)

AG = <CFG, A, CR, CC, TR>

that is composed by a context-free grammar (CFG), which has been extended to provide context by using a set of attributes; Those same set of attributes (A), which exist in each production of a grammar, and are divided into two groups, *Synthezized Attributes*, which allows values to be pass one from a node to its parent, and *Inherited Attributes*, which allows values to be pass on from the current node to a child Slonneger and Kurtz (1995); 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 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, with a class on computacional linguistics. Teachers from both classes used the same methodology to teach their classes, na dnoticed that it wasnt 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 Sierra and Fernández-Valmayor (2006).

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 Sierra and Fernández-Valmayor (2006). 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.

2.4 VisualLISA (A Visual Programming Environment for Attribute Grammars)

VisualLISA is a visual programming environment created by Pedro Oliveira in the year of 2009 Oliveira et al. (2009) 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 2.1) 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.

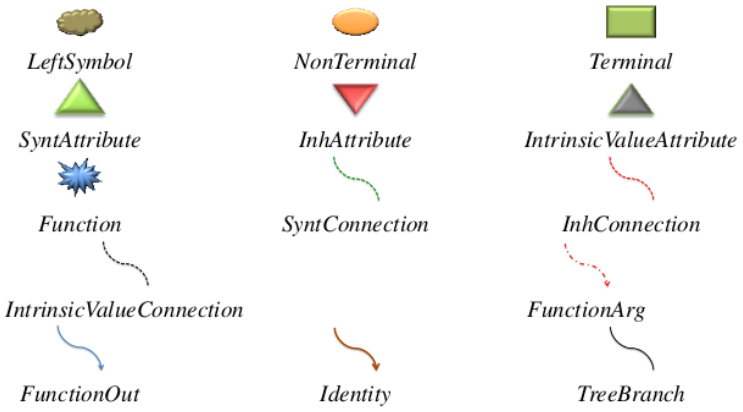


Figure 2.1: VisualLISA set of icons.

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

The environment (Figure 2.2) 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.

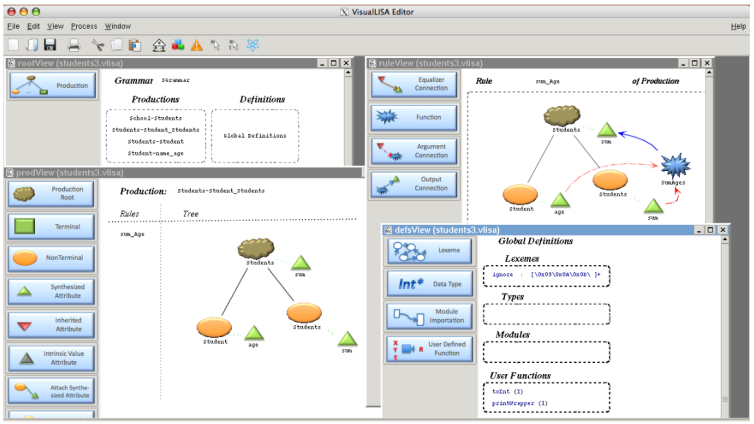


Figure 2.2: VisualLISA main window.

As an example, it was included a textual specification (Figure 2.3) and the respective graphical specification (Figure 2.4), extracted from the proper article Oliveira et al. (2009).

```
1 P1: Students → Student Students
2         {Students0.sum = Student.age + Students1.sum}
3 P2: Students → Student
4         {Students.sum = Student.age}
5 P3: Student → name age
6         {Student.age = age.value}
```

Figure 2.3: Students textual grammar.

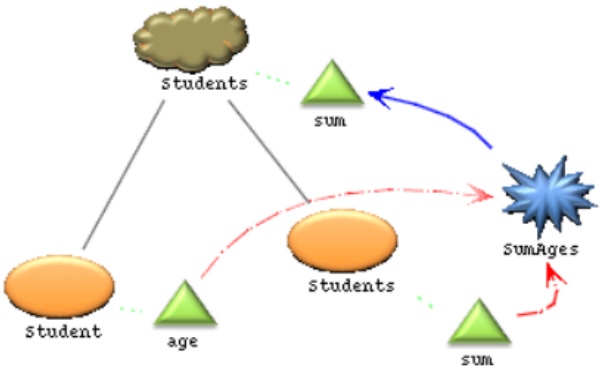


Figure 2.4: Students graphical grammar.

The main effect of the tool is not directly related to the problem that is trying to be solved, what is useful are all the visual components that are associated with it, which could help when creating the intended user interface for the visual analysis of the generated syntax-tree, or more. Nevertheless, it is a very useful and interesting way of aproaching attribute grammars and their specifications.

Chapter 3

Proposal

The main purpose of this proposal is the definition of a DSL that allows the specification of all different kinds of sentences, and afterwards, the possibility of testing given sentences as input and check if they are written accordingly to the rules previously specified. 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" is suppost to be the language in which the teacher would specify the rules for sentence construction. These rules would be written (in a single file) according to the following structure, that is divided into three main categories:

1. **STRUCTURE** - this would be the block where the teacher would

write how was the sentence suppost to be written, and what components would it have.

2. **ERRORS** - list of conditions that the teacher would write in order to be analysed afterwards, for example, certain values for different attributes. If one of the conditions was matched, an error would appear.

3. **INPUT** - this block corresponds to the "parsing" of the sentence (the lexical part) that the student wants to test. This would be written by the student and then automatically joined with the teachers information.

This file would then be processed by an ANTLR processor that would 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 would be used to create a validator of sentences, where the student could write his sentence/sentences and obtain results. These results would be the validation of the given sentence/sentences and a tree for a better visualization of the input structure.

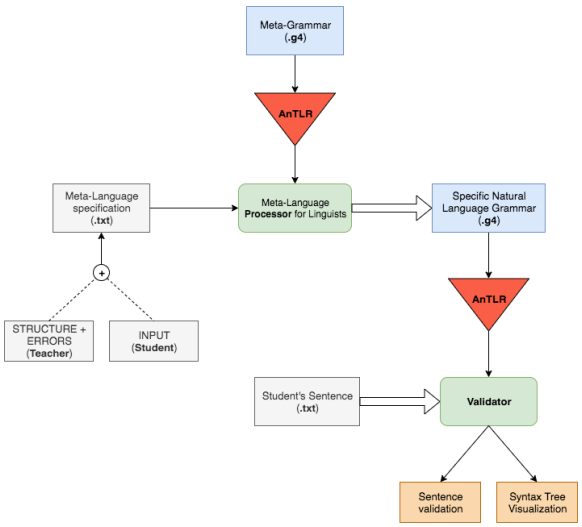


Figure 3.1: System architecture.

3.2 Meta-Language

As it was stated in the beginning of this document, the main goal was to create a DSL that was 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

```
1 processor : structure errors input
2 ;
```

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

```
1 structure : 'STRUCTURE:' (part)+ ;
2
3 part : 'part' element ;
4
5 element : '(' WORD ( ',' attributes )? ( ',' subparts )? ')'
6     ('?')? ;
```

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

```
1 attributes : 'attributes' '{' WORD ( ',' WORD )* '}'
2 ;
3
4 subparts : 'subparts' '[' element ( ',' element ) ']'
5 ;
```

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. These restrictions are logical conditions that must be obey for a sentence to be valid based on a specific structure.

Listing 3.4: DSL errors/expression productions

```
1 errors : 'ERRORS:' ( expression ';' )+
2 ;
3
4 expression : condition ( ('AND'|'OR') condition )*
5 ;
```

The **expression** production is a set of conditions that can be joined using the logical operators 'AND' and 'OR'. Each condition intends to access each attribute of any component and then assign it a value.

Listing 3.5: DSL condition production

```
1 condition : WORD ( '.' WORD )* '->' WORD
2     ('='|'!=') '"' WORD '"'
3 ;
```

If, for instance, the teacher says that an attribute is equal to some value, then the student can not use said value with said 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. By being treated as a separate DSL, it is accessible when it comes to changes, and simple to produce them.

Listing 3.6: DSL input production

```
1 input : 'INPUT:' ( '-' parts )+
2 ;
```

The sketch starts within a section named **parts**, which corresponds to one sentence in particular. This production is composed by one or more 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

```
1 parts : '(' block ( ',' block )* ')'
2 ;
3
4 block : WORD content
5 ;
6
7 content : (slice)? (attrs)? (parts)?
8 ;
```

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

```
1  slice : ':' '“' (WORD)+ '”'
2  ;
3
4  attrs : '[' evaluations ']'
5  ;
6
7  evaluations : eval ( ',' eval )*
8  ;
9
10 eval : WORD '=' '“' WORD '”'
11 ;
```

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 semantic 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,
        attributes{tipoVerb, tipoComplDir, tipoComplInd},
        subparts[
            (Verbo),
            (Complemento_Direto, subparts[(Determinante)?, (Nome)]),
            (Complemento_Indireto, subparts[(Pronome), (Quantificador)?, (Nome)])?
        ]
    )

ERRORS:
    Predicado->tipoVerb = "sujAnimado" AND Sujeito->tipo = "inanimado";
    Predicado->tipoVerb = "complAnimado" AND Predicado->tipoComplDir = "inanimado";
    Predicado->tipoVerb = "sujInanimado" AND Sujeito->tipo = "animado";
    Predicado->tipoVerb = "tempo" AND Predicado->tipoComplDir != "tempo" AND
        ↪ Predicado->tipoComplDir != "null";
    Predicado->tipoVerb = "tempo" AND Predicado->tipoComplInd != "tempo" AND
        ↪ Predicado->tipoComplInd != "null";
```

In the case of the student, this is the semantic 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"
        [tipoVerb = "sujAnimado", tipoComplDir = "tempo", tipoComplInd = "null"]
        (Verbo: "teme", Complemento_Direto: "a sinceridade"
            (Determinante: "a", Nome: "sinceridade")))
```

3.3 Case Study

After the conception of a possible system architecture and DSL, the first task was to define the DSL that was supposed to be generated by the "meta-language" processor. This DSL would be the translation of the previously defined language into ANTLR instructions, which main purpose is to verify the sentences written by the students. The DSL design consists in slicing the sentence into main parts, with each main part possibly having more parts into them. These parts are supposed to be specified by the teacher, but in this context, as a case study, it was assumed that a structure is predefined.

3.3.1 Attribute Validation

This case study intends to validate sentence components based on their attributes, where certain components require certain types of attributes in different components of the sentence. The example shows that one sentence is divided into two main parts: a subject (sujeito) and a predicate (predicado). The subject is then subdivided into a possible pronoun (pronome) and a noun (nome), which are then matched with a word (in this example the words are predefined). The predicate is composed by the verb (verbo) and two complements that are directly and indirectly (complementoDireto and complementoIndireto) connected to the verb. Both these complements are composed by a possible pronoun or determinant (determinante) followed by a quantifier (quantificador) and a noun. The verb is, like the noun, predefined.

Listing 3.11: Example of a possible sentence grammar

```
1  oracao : sujeito predicado '.' ;
2
3  sujeito : (determinante)? nome ;
4
5  predicado
6      : verbo complementoDireto
7      | verbo complementoIndireto
8      | verbo complementoDireto complementoIndireto
9  ;
10
11 verbo : '...' ;
12
13 complementoDireto : (determinante)? nome ;
14
15 complementoIndireto
16     : (pronome)? (QUANTIFICADOR)? nome ;
17
18 pronome : '...' ;
19
20 determinante : '...' ;
21
22 nome : '...' ;
```

All that is left to do is to start evaluating all the different attributes that each component has, and to check if the sentence provided by the student is, according to the rules, valid.

As shown in the excerpt below, and based on different set of attributes and rules, if the verb requires an **animated** subject then if the subject is **inanimated** an error message should appear, and notify the student of that same error. These error messages are the translation of the "ERRORS" block explained on the previous section.

Listing 3.12: Example DSL attribute validation

```
1  oracao : sujeito predicado '.'
2  {
3      if ($predicado.req_verbo == SUJEITO_ANIMADO) {
4          if ($sujeito.tipo == INANIMADO) {
5              String err = String
6                  .format(
7                      "- O verbo '%s' requer um
                           ↪ sujeito ANIMADO.\n",
8                      $predicado.verbo_TXT
9                  );
10
11             System.out.println("\nERRO: ");
12             System.out.println(err);
13         }
14     } else if { ... }
15 }
```

To conclude, if we use as an input a sentence like

"O Carlos teme a sinceridade."

the sentence is accepted because, firstly, the structure is correct, and every component is in the right place, and secondly, all the attributes obey to the rules. As another example, if we use as an input other sentence for example

"O acidente teme a sinceridade."

we get an error which indicates that the verb "teme" requires an animated subject, and "acidente" belongs to the subject that has an inanimated property.

Listing 3.13: Example of an error message

```
1 ERRO:
2 - O verbo 'teme' requer um sujeito ANIMADO.
```

3.3.2 Number and Gender

This case study, based on the same DSL excerpt [**??**], intends to validade the gender and number within the subject. Two of the productions, **determinante** and **nome**, need to be in concordance when it comes to gender and number. By using synthesized attributes, we can return the genders and numbers of the given words.

Listing 3.14: Excerpt of the "determinante" production

```
1  determinante returns[Integer genero, Integer numero]
2      : ('A' | 'a')
3          { $genero = FEMININO; $numero = SINGULAR; }
4      | ('O' | 'o')
5          { $genero = MASCULINO; $numero = SINGULAR; }
6      | ('As' | 'as')
7          { $genero = FEMININO; $numero = PLURAL; }
8      | ('Os' | 'os')
9          { $genero = MASCULINO; $numero = PLURAL; }
10
11     (...)
12 ;
```

Listing 3.15: Excerpt of the "nome" production

```
1  nome returns[Integer genero, Integer numero]
2      : 'Carlos'
3          { $genero = MASCULINO; $numero = SINGULAR; }
4      | 'sinceridade'
5          { $genero = FEMININO; $numero = SINGULAR; }
6
7      (...)
8  ;
```

Within the **sujeito** production, we can perform these two evaluations, and check if the sentence provided by the student is, again, according to the rules, valid.

Listing 3.16: Example of gender/number validation within the "sujeito" production

```
1  sujeito returns[Integer tipo]
2  @init {
3      boolean flag = false;
4  }
5      : ( determinante { flag = true; } )? nome
6      {
7          if (flag) {
8              if ($determinante.genero != $nome.genero) {
9                  String err = String
10                     .format(
11                         "- Discordancia de genero entre o determinante '%s' e o nome
                               ↪ '%s'.",
12                         $determinante.text,
13                         $nome.text
14                     );
15
16                 System.out.println("\nERRO: ");
17                 System.out.println(err);
18             }
19
20             if ($determinante.numero != $nome.numero) {
21                 String err = String
22                     .format(
23                         "- Discordancia de numero entre o determinante '%s' e o nome
                               ↪ '%s'.",
24                         $determinante.text,
25                         $nome.text
26                     );
27
28                 System.out.println("\nERRO: ");
29                 System.out.println(err);
30             }
31         }
32
33         $tipo = $nome.tipo;
34     }
35 ;
```

To conclude this case study, if we use a similar input, but with an error

"A Carlos teme a sinceridade."

we get a message indicating that the determinant and noun have a different gender.

Listing 3.17: Example of a gender error message

```
1 ERRO:
2 - Discordancia de genero entre o determinante 'A' e
    ↪ o nome 'Carlos'.
```

Using another input, for example

"Os Carlos teme a sinceridade."

we can see that the gender matches, but the number in incorrect. Likewise, we get an error.

Listing 3.18: Example of a number error message

```
1 ERRO:
2 - Discordancia de numero entre o determinante 'Os'
    ↪ e o nome 'Carlos'.
```

Chapter 4

System Workflow

This section will present the 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 expandable, allowing for certain calculations to be injected over the tree.

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 meta-grammar. After that, a single file is passed on to the processor for verification. This is the file containing the three main parts.

Within the grammar itself, the first rule

Listing 4.1: Processor rule from the meta-grammar

```
1  processor
2  @init {
3      /* Main data structure. */
4      List<RoseTree> struct = new ArrayList<>();
5
6      (...)
7  }
8      : structure[struct]
9        errors[struct]
10       input[struct]
11     {
12         (...)
13     }
```

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 their could 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

```
1  @members {
2      class Component {
3          String name;
4          String lexical_part;
5          Map<String, String> attributes;
6      }
7
8      /* Main data structure. */
9      List<Component> struct;
10 }
```

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 input, checking if it obeys to 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

```
1  @members {
2      class Graph
3      {
4          Map<Component, List<Component>> map;
5      }
6  }
```

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 in 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

```
1 class RoseTree {
2     String value;
3     boolean required;
4     Map<String, String> attributes;
5     List<String> lexical_part;
6     List<RoseTree> children;
7
8     (...)
9 }
```

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 existing), 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 ***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 that 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 a attribute ***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. When parsing these segments, they are stored within a node of the *Rose Tree*, to then be injected into the

generated grammar. In the case of any error, the user is informed of where the error happened but also in which component. Furthermore, when parsing attributes and their respective values, if the student defines two attributes with the same name, but with different values, a warning is raised to inform the user that the value that was considered was the last one to be recognized.

Having parsed the meta-language file, a generator is called by the main rule - the *Rose Tree* is passed as an argument and then traversed in order to generate all the rules for the grammar. This grammar, after being generated, is used to create a processor in which the student's sentence will be used as input, creating a visual syntax tree of that same sentence.

Chapter 5

Conclusion

With this document, the main objective was to conduct a study and analysis about what this problem involves and what could, in any way, help create an adequate solution. Furthermore, the first approach to the problem was documented, giving special attention to the case studies and a first sketch of what will become a new language for linguists.

Regarding to the approaches that were taken, it is quite clear that some parts are still at an early stage of development, and require more time - mainly the design of the students DSL, which is still very verbose in comparison to what was projected. The next step, which was already taken, is the creation of the meta-grammar that processes the information written by the teacher + student, and generates the ANTLR instructions based on the defined structure. The generated grammar will be based on the DSL structure that was used for the case studies. Afterwards, with a functional validator, the goal is to build a system with a user interface that allows to visualize the syntax tree of the input sentence, helping when it comes to analyse each segment individually.

5.1 Working Plan

The outlined work plan for this master thesis will consist of six phases. Each phase will include the conclusions of the previous phases and build upon the knowledge gained in each one.

Phase 1 Bibliographic search in applying attribute grammars to linguistics, and study the tools that are already available.

Phase 2 Design the domain specific language (DSL) with all the requerements previously mentioned.

Phase 3 Create the language translator in ANTLR.

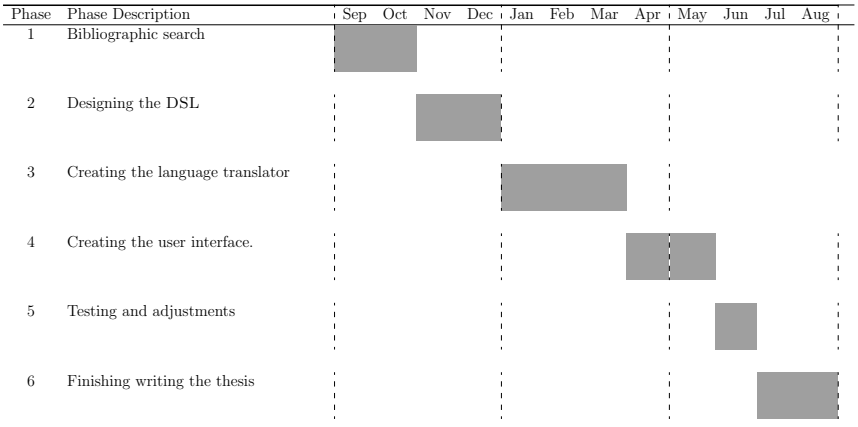
Phase 4 Create the user interface.

Phase 5 Test and required adjustments.

Phase 6 Write Thesis.

For a better visualization, it was created a diagram based on all the different phases, divided into the months.

Table 5.1: Activities Plan detailed



Bibliography

Barros, P. A., Varanda Pereira, M. J., and Henriques, P. R. (2017). Applying attribute grammars to teach linguistic rules. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

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

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

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

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

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

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

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

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

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

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