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

January 5, 2021

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

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 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 Domain Specific Language

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 van Deursen et al. (2000).

This type of languages provides 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; 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 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, 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 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 Nuno 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.

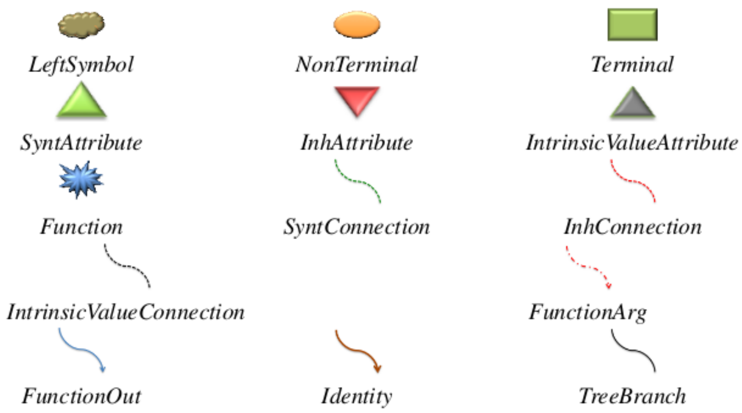


Figure 2.1: VisualLISA set of icons.

[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.

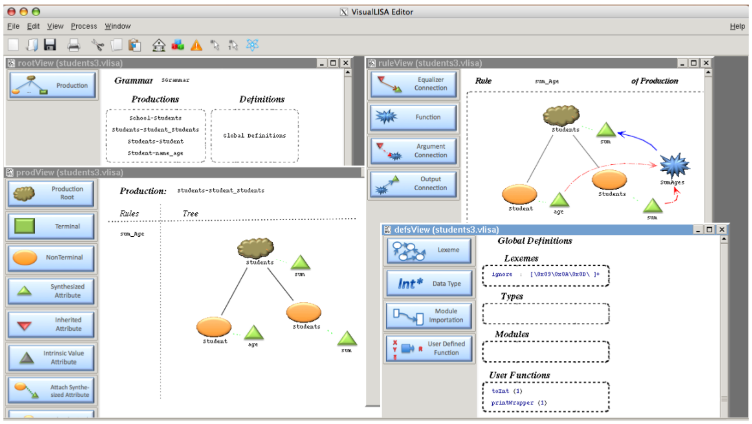


Figure 2.2: VisualLISA main window.

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

```
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 2.3: Students textual grammar.

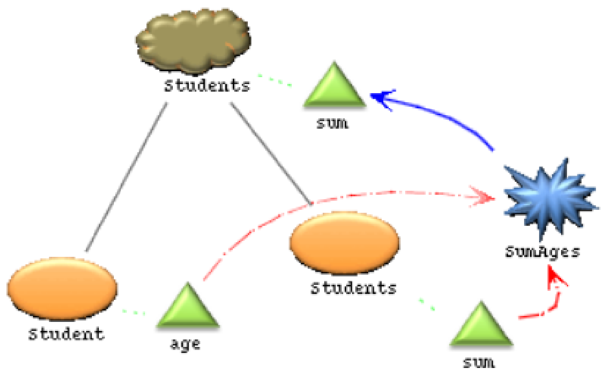


Figure 2.4: Students graphical grammar.

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" 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** - list of conditions that the teacher could write in order to be analysed afterwards, for example, certain values for different attributes. If one of the conditions is matched, an error will appear.

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.

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. These results would be the validation of the given sentence/sentences and a tree for a better visualization of the input structure.

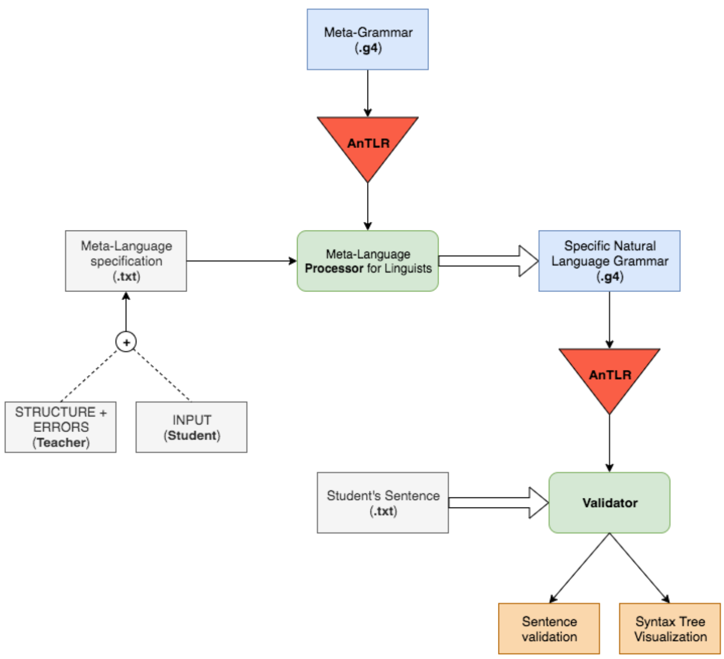


Figure 3.1: System architecture.

3.2 Meta-Language

As it was stated in the beginning of this document, the main goal was to create a DSL that 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 ( ',' 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 : 'ERRORS:' ( expression ';' )+
;

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

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

```
condition : WORD ( '.' WORD )* '->' WORD
    ('='|'!=') '“' WORD '”'
;
```

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

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

```
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”)))
```

Chapter 4

System Development

This section 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.

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 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

```
@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 $\boldsymbol{A}$, and this component has two children, $\boldsymbol{B}$ and $\boldsymbol{C}$, then the paths $\boldsymbol{A} \rightarrow \boldsymbol{B}$ and $\boldsymbol{A} \rightarrow \boldsymbol{C}$ 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 $\boldsymbol{N}$.

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 $\boldsymbol{N}$ 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

```java
class RoseTree {
    String value;
    boolean required;
    Map<String, String> attributes;
    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.

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.

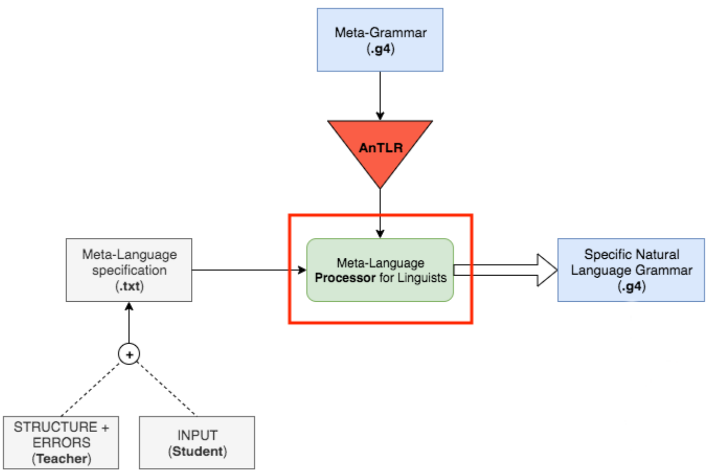


Figure 4.1: Excerpt of the system architecture.

4.3 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 once again the command line, and providing a specific flag to the tool (-gui):

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

we obtain the final syntax tree for the sentence provided.

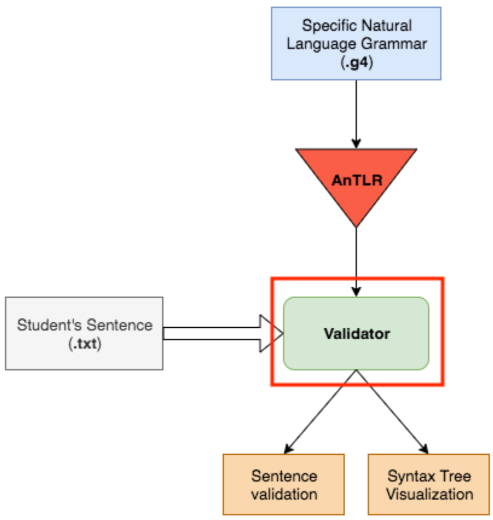


Figure 4.2: Excerpt of the system architecture.

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

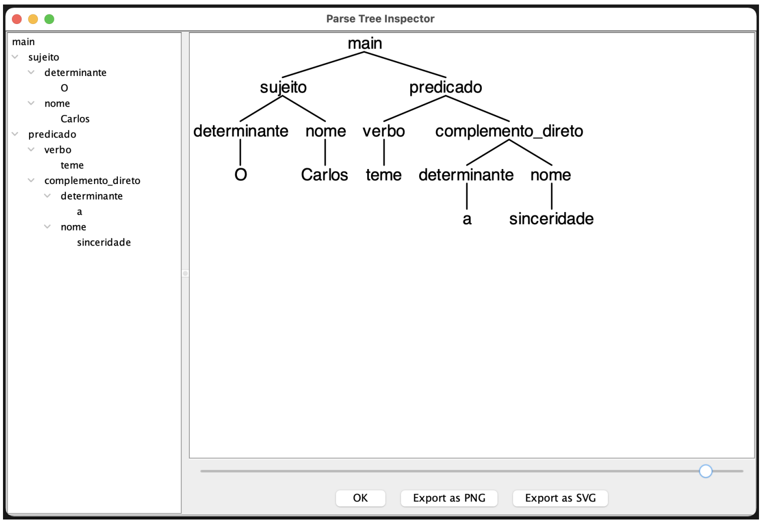


Figure 4.3: Example of a generated syntax tree within TestRig.

4.4 User 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 *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 and INPUT blocks directly into the respective text areas without opening any file.

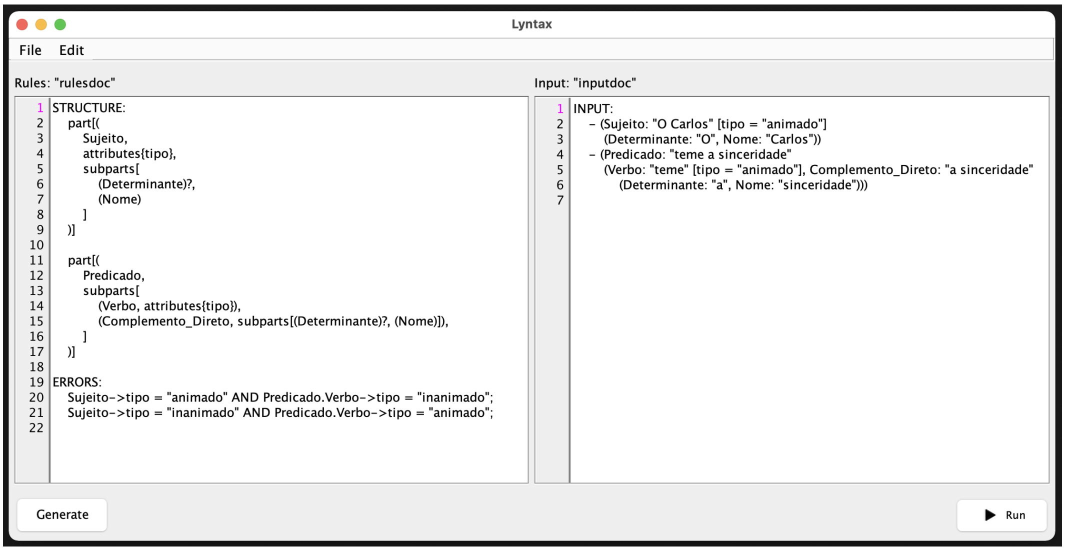


Figure 4.4: Lyntax user interface.

After the specification of the rules and input, 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.

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 know 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.

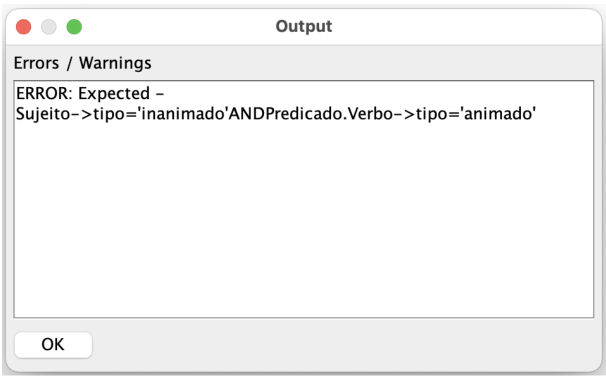


Figure 4.6: Example of an error message.

Chapter 5

Case Studies

In order to better explain the solution architecture presented in this thesis, 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.

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. 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 [??] 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 pronoun (Pronome) 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 determinant (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 evaluate if each of the components are animated or inanimated. By analysing the logic in the ERRORS block [??], 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 [?? plus ??] 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 */
(...)
```

In 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 [??], 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 [**??**] 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 [**??**]). 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 */
(...)
```

This simple 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 simple (but concrete) and complex sentences 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 more common tested structures and to touch on the topic of Complex Sentences, showing that it is certainly possible to have them translated into the meta-language.

This first example is, again, composed by a Subject - Predicate structure, with some caracteristics to the Predicate itself.

Listing 5.11: Example of a sentence structure

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

Listing 5.12: 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*).

Listing 5.13: 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 5.14: 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 5.15: Example of a sentence structure

```
STRUCTURE:
    part[(
        Frase, subparts[
            (Interjeccion | Pronombre),
            (Verbo),
            (Nombre)
        ]
    )]
```

Listing 5.16: Example of a sentence input

```
INPUT:
    - (Frase (
        Interjeccion: “Hola!”,
        Verbo: “Soy”,
        Nombre: “Manuel”
    ))
```

Complex Sentence...

Listing 5.17: Example of a complex sentence structure

```
STRUCTURE:
    part[(Sujeito)]
    part[(
        Predicado,
        subparts[
            (Verbo),
            (Complemento_Indireto)
        ])
    )]
    part[(Conjuncao)]
        part[(
        Predicado,
        subparts[
            (Verbo),
            (Modificador_Verbal)
        ]
    )]
```

Listing 5.18: Example of a complex sentence input

```
INPUT:
    – (Sujeito: "Los soldados")

    – (Predicado (
        Verbo: "dispararam",
        Complemento_Indireto: "a los sentenciados"
    ))

    – (Conjuncao: "y")

    – (Predicado (
        Verbo: "cayeron",
        Modificador_Verbal: "muertos"
    ))
```

Chapter 6

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.

6.1 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 requeriments previously mentioned.

Phase 3 Create the language translator in ANTLR.

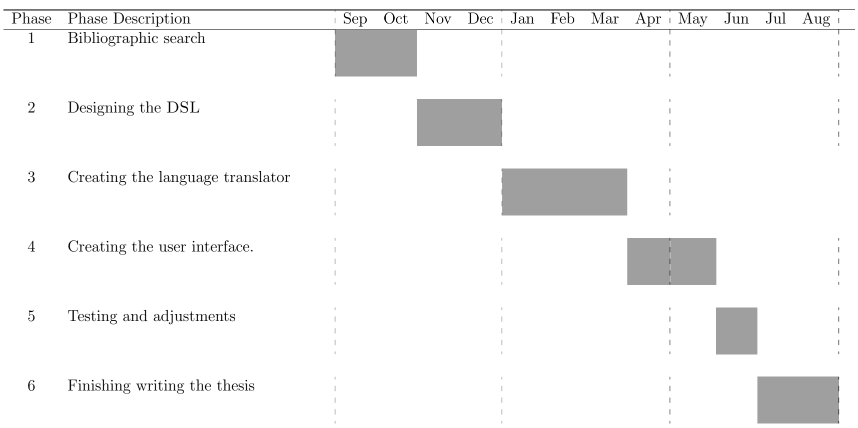
Phase 4 Create the user interface.

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 6.1: 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.