Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

João Carlos Alves Cruz

QGE - An Attribute Grammar based System to assess Grammars Quality

Master dissertation

Supervised by: Pedro Rangel Henriques

Co-supervised by: Daniela da Cruz

Braga, October 27, 2015

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

João Carlos Alves Cruz

QGE - An Attribute Grammar based System to assess Grammars Quality

Master dissertation

Supervised by: Pedro Rangel Henriques

Co-supervised by: Daniela da Cruz

Braga, October 27, 2015

PARECER

Serve o presente parecer para declarar que o aluno João Carlos Alves Cruz concluiu, conforme esperado, a escrita do seu relatório de dissertação. O documento foi revisto pelos orientadores, os quais atestam a sua validade científica, assim como o cumprimento dos objetivos propostos para esta etapa.

Mais se informa que as atividades de mestrado do aluno João Carlos Alves Cruz decorrem dentro dos planos e prazos inicialmente previstos.

ORIENTADOR Pedro Rangel Henriques

CO-ORIENTADOR Daniela da Cruz

ACKNOWLEDGEMENTS

ABSTRACT

The development and support of a software system is a complex task, therefore software engineering is responsible to supply a collection of techniques and scientific methods, to deal and cope with such complexity. Analysis, modeling/specification, design, maintenance, among others, are activities included in this field of engineering.

This master work aims at proposing an approach for the application of these techniques and methods within languages, more precisiley, grammars. Emphasis is put on the grammars development process, implementation and maintenance, aiming for better results, in terms of efficiency and quality, concerning both the generated language and its processor. Assuming a grammar, gaps were identified related to the study of its quality, more specific, in terms of ease of learning (reading and understanding), ease of derivation and maintenance, as well as in terms of efficiency, like the language sentences recognition and the generation of the language processor. A set of metrics is proposed to overcome those gaps and improve the quality of grammars.

This process is based on: *i*) selecting a set of well-defined software metrics, for the quantitative measurement of the quality of grammars; *ii*) deciding how each metric relates to the characteristics which affect the quality of a grammar.

The solution that will be presented here aims at developing a tool, based on Attribute Grammars to assess, in an automatic way, the quality of grammars. The implementation of the grammars quantification process enables the achievement and discussion of results for a given attribute grammar as well, as it allows us to take elations concerning the language that is defined by this grammar, due to a symbiotic relationship between the two concepts . Another important aspect, after reading the results obtained by this process is the direct manipulation of grammars, turning them into a higher quality software product, validated by well-defined software engineering techniques.

Keywords: grammar engineering, attribute grammars, software metrics, software quality and maintenance

RESUMO

A construção e o suporte de um sistema de software é um processo complexo, pelo que a engenharia de software é responsável por proporcionar uma coleção de técnicas e métodos científicos, de acordo com essa complexidade. Inclui atividades como a análise, modelação/especificação, design, implementação, manutenção, entre outros.

Neste documento de tese de mestrado propõe-se uma abordagem de aplicação destas técnicas e métodos no âmbito das linguagens, mais precisamente, das gramáticas, num ramo da engenharia de software conhecido por Engenharia Gramatical. A proposta foca-se no processo de desenvolvimento, implementação e manutenção de gramáticas com o intuito de melhorar os resultados em termos de eficiência e qualidade, quer da linguagem definida pela gramática, quer do respetivo processador. Pesquisando no universo da engenharia gramatical, identificaram-se lacunas referentes a estudos sobre a sua qualidade, mais propriamente, em termos de facilidade de aprendizagem (leitura e compreensão), facilidade de derivação e manutenção, bem como em termos de eficiência, tanto no reconhecimento de frases da linguagem como na geração de um processador para essa linguagem, e propõe-se uma solução para superar tais lacunas, através de um processo de avaliação baseado em medidas rigorosas de certos parâmetros.

Este processo baseia-se: *i*) na seleção de um conjunto de métricas de software bem definidas, para a aferição quantitativa da qualidade das gramáticas; *ii*) na escolha da forma como cada métrica se relaciona com as características que influenciam a qualidade de uma gramática.

A solução que será aqui apresentada visa o desenvolvimento de uma ferramenta, baseada em Gramáticas de Atributos para aferir, de uma forma automática, a qualidade das gramáticas. A implementação do processo de avaliação quantitativa de gramáticas possibilita a obtenção e discussão de resultados referentes a uma dada gramática de atributos, bem como, permite tirar elações referentes à linguagem que é definida por essa gramática, devido a uma relação simbiótica que existe entre os dois conceitos. Outro aspecto importante, depois de uma leitura aos resultados obtidos por este processo é a manipulação direta das gramáticas, transformando-as num produto de software de maior qualidade, validado por técnicas de engenharia de software bem definidas.

Palavras-Chave: engenharia gramatical, gramáticas de atributos, métricas de software, qualidade de software

CONTENTS

LIST OF FIGURES

List of Figures

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

1 INTRODUCTION

As is well known currently the human society is crossing the Information Age or Digital Age, where the paradigm dictates the application of digital technology in all aspects of our lives, such as individual business, governments, mass communications, art, medicine or science. Although the term engineering have emerged in ancient times as a way to apply mathematical, technical and scientific knowledge for the creation, maintenance and improvement of materials, devices, structures, systems or processes, we continue to see today in the *software field*, news of software errors with tragic consequences.

Just to give a veridical example among many others, somewhere between 1985 and 1987, a radiation therapy machine called Therac-25, was involved in a catastrophic accident killing at least six patients, due to a software error. This error caused the management to patients of radiation doses that were thousands of times greater than normal, resulting in death or serious injury[1].

To avoid that accidents like the one mentioned before, it is necessary to improve the quality of software products. It is precisely in this context that this master's thesis fits, but addressed to the grammatical software field of study.

1.1 GRAMMAR ENGINEERING

Grammar Engineering [Klint et al. (2005), Alves and Visser (2008), Lämmel (2001), Erbach (1992)] is a field in software engineering that involves the application of well studied software techniques and methods to grammar, just as they are applied on another software product. Such techniques include version control, static analysis, unit testing, grammar software metrics, grammar evolution, refactoring among others. Through their implementation, in today's process of developing and maintaining large grammars, better results can be achieved in terms of quality, increasing their efficiency and confidence[2].

Grammars are present everywhere in software development and their use has been growing in importance assuming them as tools for the definition of concrete or abstract programming languages syntax, exchange formats in component-based software applications and others. The software industry is faced with various kinds of applications that rely on grammars such as compilers, debuggers,

1 For more information and other examples consult http://royal.pingdom.com/2009/03/19/10-historical-software-bugs-with-extreme-consequences/
2 In terms of processing and results.

slicing tools, documentation tools, language tools, reference manuals, browsers and IDE's, software analysis tools, code processing tools and software transformation tools. So, it is expected that a disciplined engineering plays a relevant role, but that is not the case.

Klint, Lammel and Verhoef [Klint et al. (2005)] introduced the concept of *Grammarware*, to represent all the software systems which core relys on grammars. In other words, software that involves or encodes grammatical structure. This kind of software provide a critic level of automated support among software development life cycle. Looking to grammars as formal specifications, profits can be taken by using that formal approach to analyze and assess them.

1.2 MOTIVATION

As happens in human society, when two or more people wants to communicate, it is necessary to define a **Language**[3], creating cooperation and co-existence. This language is the key for the interaction between humans as is for the interaction between a person and a computer or machine [Henriques (2013)]. Therefore, the language has an important role in the communication process because must ensure that the sender knows exactly what he wants to transmit and the receiver understands exactly what was submitted. To accomplish that task correctly and in an efficient way, the language must have some kind of support, that indicates how to transmit valid sentences with the right semantics and how to interpret that sentences for the right extraction of their meaning, and that is where **Grammars**[4] come in.

This two important processes, writing the sentences and executing their recognition, are within the scope of grammar processing, in the way they define the language. The *relation* that exists between a grammar and the language that is defined by that grammar is undeniable. This relationship allows for the discussion that will be addressed in this document: from the results obtained by evaluating the quality of a grammar, reasoning can be made and elations can be taken about the quality of the language that is defined by that grammar. It will be proposed that grammars can be, not only a tool that defines a language and its processing, but also a tool for its evaluation [Henriques (2013)].

Software Engineering aims to put together methods and processes that allows for a effective development of software systems. The same can be told about grammar engineering and the development and maintenance of large scale grammars[5], as a software system. The need for refinement of the quality of grammars, both in its development and in its maintenance, arising from gaps in grammar engineering may be briefly identified:

- The techniques and methods of software engineering are used since 1960[Kan (2002)] and yet play an important role in today's software development. After the functional era, the schedule era and the cost era, we now stand on the quality and efficiency era. Because of the increasing of

3 Something simple such as a set of sentences (sequence of symbols from an alphabet).
4 In real life as in the software development
5 In this context all types of grammars are considered, such as Context Free Grammars, Attribute Grammars, Sintax Definition Formalism Grammars [Alves and Visser (2005)], and others.

software products in every aspects of our lives, it became necessary to reason about things that were not considered before[6]. The problem is that this principles and fundamentals of software engineering are not being applied, in a sustained and efficient way, to the development and maintenance of grammar-dependent software[7][Klint et al. (2005), Power and Malloy (2004), Bender (2007), Lämmel (2001)].

Grammars cannot be regarded as formal aspects such as parsing algorithms. The way to go is paying more attention to the engineering aspects of grammars and all of grammar-based software, applying to them the same techniques and methods that are applied to other software products, leading to the improvement of quality of grammarware and to increase the productivity of grammar-based software development.

> *In reality, grammarware is treated, to a large extent, in an ad hoc manner with regard to design, implementation, transformation, recovery, testing, etc.* ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 3, July 2005.

- The act of writing a program is an activity that has to be tested and modified until it reach the expected behavior. Because programming is a difficult process, the good-practice rules dictate that this process cannot be done in a fast, instinctive and without thinking way. Well, the writing of large grammars follow the same pattern but in a incremental way, where a cycle of: writing or modifying the grammar, testing the grammar, debugging the grammar, should be adopted. Grammar engineering tools must support this cycle[Erbach (1992)].

- This deficiency of engineering related to grammars is noted also in terms of actual costs, especially in companies that depend crucially on grammars or grammar-based software. Complex grammars can be difficult to understand and therefore difficult to maintain. The maintenance process required is a big slice of the software budget [8].

  Costs related to software quality are very important and the reference to technical debt[de Groot et al. (2012)] is often made. This charge represents costs in repairing problems in software systems to achieve the ideal quality. The interest in debt represents additional costs to maintain a software system due to its *lack of quality*. The proposal made in this document is setup in this context, where Quality is assessed from engineering principles.

In the recent development of grammar-based software, Attribute Grammars[9] have been a very often used tool in many products including compilers construction, detection of program anomalies, as a basis for a language-based editor, and a basis for a software development paradigm [Ghani and Hunter (1996)], for the specification of the programming language.

6 Such as verification and certification of software, when there exists lives at stake, for example.
7 In grammar engineering the term *grammar hacking* is often used to describe precisely this situation where issues like testing and disciplined adaptation of grammars play a minor role.
8 Estimates of the cost of maintenance range from 65% to 80% of the total software budget.
9 A Context-free Grammar with a set of attributes associated. Each attribute represents a static semantic property of the associated symbol. A more detailed definition is given ahead on this document, in the State Of the Art chapter.

Assuming the premise that with the expected results of higher quality, it is possible to develop more robust and reliable grammar software, is a strong motivational factor could lecture on something that is used a lot in the development of modern software and we can introduce principles of engineering to make it better. This is the essence of a software engineer.

1.3 AIMS/OBJECTIVES

The aim broader of this master work consists in providing to software engineers and others developers in the language processing field, a tool to evaluate the quality of the Attribute Grammars automatically.

Based on a set of well established software metrics, to assess the quality of the software, a mapping is done from this general software metrics to be applied to grammar. Then, with this set of grammar metrics, the tool accept an Attribute Grammar as input to automatically assess the quality of that grammar.

In a more specific sense, objectives to be achieved can be described as follows:

- The development of a grammar-based software tool, as described in the first and second paragraphs of this section, with the following features:
  - allowing to define the metrics to calculate, linking the synthesized attributes[10] to the symbols of the grammar;
  - allowing to define rules to evaluate the attributes, according to the meaning of its metric;
  - accepting a given grammar, and assess their quality evaluating the attributes;
  - allowing grammar manipulation, for example transforming into another equivalent grammar with better quality.
- Provide techniques, well-founded and proven methods to support the development, maintenance, recovery and the implementation of attribute grammars;
- Supervising and controlling the development process of attributes grammar, improving aspects such as performance, reliability and efficiency, always in a well sustained and scalable manner according to the complexity of each grammar presented, leading to better quality software systems;

In the long term, and as a result of work performed it is desired to, reduce the cost of maintenance of grammar software.

With the objectives presented above, the final outcome intended is one reasoning platform to apply the most rigorous software process and provide a great support to grammarware and more precisely to attribute grammars.

10 Shortly, they represent the transmission of semantic information in a *bottom-up* direction, on the syntactic tree. In the opposite *top-down* direction, inherited attributes are used. More ahead in this document, this theme will be detailed.

1.4 DOCUMENT STRUCTURE

The present document is organized in six chapters that describe the working area and exposes a proposal for assessing the quality of grammars, more precisely Context-Free Grammars and Attribute Grammar, and as consequence the quality of the Languages specified by those grammars.

The first Chapter present the field of study in which this theme is involved, the context , the motivation that led to this work and the main goals or aims appointed. It is desired that questions like *What for the purpose of?* and *Why bother with?*, are proper clarified.

The second Chapter addresses the State of the Art revealing the scientific method - Measurement - and formalizing the concepts involved such as the Languages Quality and the Grammars Quality. Some important formal definitions are also exposed, with the intention of clarifying the technical speech that will be used in the following chapters. The quality characteristics to be covered by the proposed software metrics for grammars, are described. In this case, the question that is intended to answer is *How it will be carried out?*

The third Chapter intents to cover the characterization of the Problem and its challenges responding to the question *What will be done?* It states, in objective terms, what should be achieved or the requirements of the solution, as well as, the execution flow for desired software system.

The fourth Chapter introduces the solution to the problem characterized in the previous chapter, in this case, a tool for assessing grammars quality named **GQE**, which stands for *Grammar Quality Evaluator*. The content of this chapter specify what kind of tool is GQE, how it was assembled, in architectural terms and the technologies involved in the development process. A brief description is made for each one of the four components, in which this software system is divided. The final section exposes the execution flow of this system, meaning what happens in the background, since the moment that the User execute the application until the moment that the results are displayed, in this case, some elations about the quality of the grammar.

The fifth Chapter shows the practical results achieved by GQE, with the help of some case study grammars. The aim is to present the different results and behaviors of this system when he is confronted with different type of grammars, in terms of quality and characteristics.

The final Chapter present the conclusions, where a summary of all document is done, relating each aspect of the thesis with the proposed objectives. A final critical reflection, upon the work that was developed, is made, pointing out some aspects of the future work that can be done in the following of this dissertation.

2 STATE OF THE ART

As all engineering, the software engineering is not limited only by the creation process of new products, but must also embrace a production with demanding requirements in terms of quality and management. Therefore quality can be assessed by different methods such as debugging, testing, assessments, risk based quality improvement and measurement. A vast literature can be found on software measurement and the state of the practice is evolving[Shoemaker and Mead (2013)].

The first section of this chapter is addressed to the foundations of measurement, What is it? and Why we do it?, followed by an introduction to the underlying theory.

The second section will cover the Languages Quality, by identifying their quality characteristics. In an ideal world this quality should allow to measure the level of satisfaction with that language serves the purpose for which it was designed, or at least allow compares several alternatives. For this document the intention is, to initially present a language quality definition and then discuss the criteria to assess language quality according to that definition, and then to relate the grammar quality with the quality of the language defined by that grammar.

The last section starts to formally define the object of study of this thesis - Grammars, then using the same procedure explained above for identifying quality characteristics on grammars[1] and ending the chapter defining a set of metrics that will be used to measure those characteristics.

2.1 THE ART OF MEASUREMENT

Measurement is an old good practice not only in software engineering [Fenton and Pfleeger (1998), Kan (2002), Kitchenham (1996)], but also in many systems in every day life. All areas of our daily lives are confronted with this concept of measurement: economics, medicine, education, construction, science, and so on. To give some more specific examples: in geology, the age of the rocks is estimated based on measurements between the parent element and the child element; in the finance business, the whole exercise of the stock market is done by measurements; measurements in radar systems enable us to detect paths or objects under the sea; architects and civil engineers use the measurements to know the quantities of a certain material to apply while building a structure; all the diagnostic systems for persons, cars, weather, etc., rely on measurement.

1 In this case Context-Free Grammars and Attribute Grammars

Many authors define measurement formally, such as [Fenton and Pfleeger (1998)]:

> *Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.*

Reasoning about the previous statement, three important concepts are highlighted: entities, attributes and rules. With measurement, information about the attributes of entities is extracted. An *entity* is something that we can compare and describe through its properties, like an object or event in the real world[2]. *Attributes* represent a feature, a property or a characteristic of entities, such as the area or the length (of a football field), the color (of a sweater) or the duration (of a battery). The *rules* or the scales, allow us to relate and compare the values of attributes between entities.

This of course is not something objective and not easy to define because it is open to a subjective interpretation. Are we measuring the characteristic or attribute that really matters? How can we validate? How can we analyze the results? Mistakes are often made, for example as in ”It is cold today” when we really mean that the air temperature is cold today, and when can we say it is cold, is there a threshold? How do we calculate it? To answer these questions we have to rely in some scientific basis and apply them to our problems.

Progress is made in terms of understanding what we measure, every time we successfully measure something that was initially unmeasurable, increasing the power of software engineering as is done in other engineering disciplines. Despite the opinion of some software engineers [Fenton and Pfleeger (1998)], that attributes like quality [3], dependability, usability and others are not quantifiable, the simple fact of proposing a set of metrics allow us to try use measurement to increase our understanding of them. In this context arise confusion between two kinds of quantification: measurement and calculation. The difference between this two terms is that we refer to **measurement** as a direct quantification or a direct assessment for gathering information about how the product behaves; on the other side **calculation** is indirect, when measures are combined into a quantified item that reflects some attribute. For example, the color of a ball or the weight of a computer is a direct quantification [4], but the amount of taxes that a family must pay is calculated using a formula that combines the number of household members, the incoming profits, the out coming expenses, and all the other factors.

2.1.1 *Measurement in Software Engineering*

Now that a short introduction to the notion of measurement in everyday life was made, let us reason about how we can apply this discipline in software engineering, more precisely to software products or processes. An engineering approach is needed, in a scientific way, to control and understand software,

2 Different types of entities can be identified such as products, processes, resources, artifacts, activities, agents, organizations, environments and constraints. An entity can also be represented as a set of other entities.

3 Precisely the attribute that this master work is based on, for grammar assessment.

4 Tell us something about the product without never have seen it. By the weight of the computer or its dimensions, I know if that computer fits on my bag, for example.

aiming for that no surprises occur in its specification, design and maintenance. The importance of software engineering can not be ignored, because software pervades our lives[Fenton and Pfleeger (1998)][5],further more it affects the quality of life.

Although faced with a significant increase in corporate[6] and institutional customers that offer this type of activity in a software product, the negligence measurement in software engineering is still present. The small amount of engineers or managers that rely on measurements do it on a incomplete, infrequent and inconsistent way[7], but the bigger slice of development projects:

- Fail to set measurable targets, the goal defined on the specification phase, for example the reliability or the usability, cannot be proven achieved, in other words, the results cannot be validated;
- The component costs of software projects are not quantified and understood; as consequence, if the project have financial losses there is no way to control it. Even if the company wants to optimize their development process to improve the incoming, they don't know how much the coding phase costs;
- The quality of the product is not quantified or predicted. Therefore, even if a company develop a good product[8], they lose the market because the quality of the product was not measured and the potential costumers cannot compare its quality with the other software candidates;
- The promotional process of the software product is done based on an misleading evidence to convince the users to buy it. Its features, advantages or a innovation technology, are not supported with scientific basis for the claims.

The direction to combat this poor and misleading process in software engineering is using measurement in a correct and valid way. This technique of measurement must be motivated by a specific and particular purpose or need, for the product or process, something that is easy to understand and clearly defined. Even for a problem free project, this is needed at least for assessing his current situation, to have the minimum control, in which state their products are, how are their processes and how useful really are their resources.

You cannot control what you cannot measure. DeMarco, 1982

Software **quality** can be measured, by discovering the faults, failures and changes as they occur, enabling us to compare, to predict the effects of changes and to assess the effects of new practices leading to product improvement.

5 A wide list of examples can be given such as airbags control, bank transactions, sophisticated medical tools, traffic control, spaceship control, sophisticated power plants and further more.
6 Just point two sample companies using the software measurement as product: the Software Improvement Group (http://www.sig.eu/en) and the Cost Engineering (http://www.costengineering.eu/index.php).
7 A good example of this is the misleading measurement in software products promotions, as illustrated on the figure.
8 In technical terms.

Our new application is 100% reliable and we can ensure a 200% improvement in your companie productivity!!!

Figure 1.: Measurement for promotion

Raising the speech to formalisms, also required, it becomes fundamental to point out the reasons [Fenton and Pfleeger (1998) and Park et al. (1996)] why it is important to measure in software:

TO UNDERSTAND As result of measure the attribute or property of a particular software product or process, we characterize to gain understanding of what is happening. Baselines are established to differ the assessment in the product life cycle as well as to set goals for future behavior. Aspects of process and product are highlighted and we can reason about what affect the entities.

TO PREDICT Getting to know the relationships between processes and product leads to an understanding which in turn leads to building models of this set of relations. With this models we can predict behavior, by analyzing the values of the attributes that were measured. Prediction allows us to set goals for the future of the projects in terms of costs, schedule and quality;

TO CONTROL At this point if we understand and predict, we are able to know exactly what is happening in projects and control them. Estimations can be done in what is more likely to happen if some change is made, what will be affected and in pejorative terms how can we avoid it. Basically, the current status of the project is clearly visible by managers and engineers and the achievement of quality goals can be assessed;

TO IMPROVE The improvement benefit is easily sighted, because measurement could be or is an iterative process. By analyzing and reasoning about the values and the results obtained in previous measures, combining the understanding and control of the processes and projects, we can always improve the properties of the measured products. The identification of deficiencies, breaking points, flaws and inadequacies opens doors for improving product quality and performance.

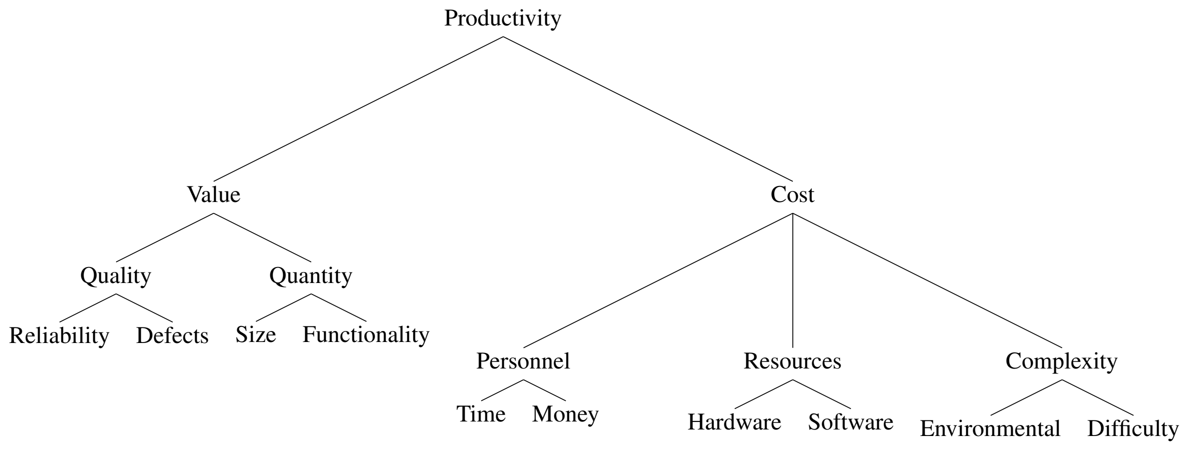


Figure 2.: Example aspects of a productivity model

2.1.2 *The Scope of Metrics*

Software measurement is presented in some portion in many operations that are supported with **Software Metrics**. All these activities, that will be listed below[9], are related to each other not only by software metrics but also by the theoretical foundations, that will be briefly addressed in the next section.

Cost and effort estimation

Many models for software cost and effort estimation have been proposed and used, as soon as the need arose to predict the costs in which and every one of the development phases, during a project lifecycle. Four of the most popular algorithmic models used to estimate software costs are COCOMO model (*Boehm, 1981*), SLIM model (*Putnam, 1978*), Function Points models (*Albrecht, 1979*) and ESTIMACS [Kemerer (1987)].

These models are used to compare the software parameters estimated against the actual values. A relation between the model and its accuracy can be spotted, so that estimators analyze it and improve accuracy for future projects.

Productivity models and measures

Productivity models and measures are used to assess staff productivity along differents stages of software development. The next figure shows an example of the possible components that affect productivity. Productivity can be assessed in this case, as a combination between value and cost, which in turn can be decomposed in other aspects, expressed in measurable form.

9 The specialists do not have a unanimous opinion as regards the separation of different types of activities. Some argue, for example, that factors such as maintenance, usability or testability should not be included in the quality models and measures, but should be addressed in a more detailed and extensively leading to an independent expertise.

Data Collection

The data collection of a program, though not appear to, have an influence in its quality. Good practices of software engineering indicate that metrics of data collection must be planned and executed in a careful and sensitive manner. Increasingly in software development, specialist evolve to achieve goals in this field: measures are defined unambiguously, that collection is consistent and complete, and that data integrity is not at risk.

The data collected allows managers to see and predict the progress and problems of development, for example, through simple charts of aspects in development, such as the time and the number of people involved in the design process, the costs of resources over time or the number of faults in the code phase.

Quality models and measures

The productivity models discussed earlier turned out to be insufficient to the production rate, since the product quality was not being assessed. Therefore thus became necessary to build quality models to combine with aspects of productivity.

The amount of theoretical materials available on this topic is vast [Fenton and Pfleeger (1998), Heitlager et al. (2007), Jørgensen (1999)] and all are based on three models of software quality: Boehm's cost estimation model is linked to a quality model (*Boehm, 1978*); McCall quality model (*McCall, 1977*); and ISO/IEC 9126-1[10] quality model. All these models are based on a methodology according which important high-level quality factors or attributes of software products are defined and composed with one or more low-level criteria (a way of measurement) for easy understanding and measurement.

The first two models are very similar in structure and propose [Cavano and McCall (1978), Boehm et al. (1978)], even some quality factor are repeated such as: usability, portability, efficiency and reliability. The latest and most fashionable ISO/IEC 9126-1[11] stresses the quality concept which includes 6 main characteristics: functionality, reliability, usability, efficiency, maintainability and portability. These attributes are divided into sub-characteristics or criteria, as illustrated in Figure 2. This quality model distinguishes three different points of view in software product quality: *i*) Internal quality for measuring properties of the system with no execution; *ii*) External quality for measuring properties in the execution moment; *iii*) Quality in use for measuring the properties experienced by its users.

Reliability models

Many of the factors mentioned in the quality models give rise to new specialization models, such as Maintainability[Heitlager et al. (2007)] models and Reliability models (*Littlewood, 1988*). There is a

10 International Organization for Standardization(http://www.iso.org/iso/home.html) standard for quality software product, which is covered by the quality model of the 9000 family of standards

11 The ISO/IEC 9126 is composed for one International Standart that defines the quality model *IS 9126-1* , and three Technical Reports: the *TR 9126-2* for External Metrics, the *TR 9126-3* for Internal Metrics and the *TR 9126-4* for Quality in Use Metrics. This three reports list a consensual inventory of metrics for assessing the quality factors defined in the models.

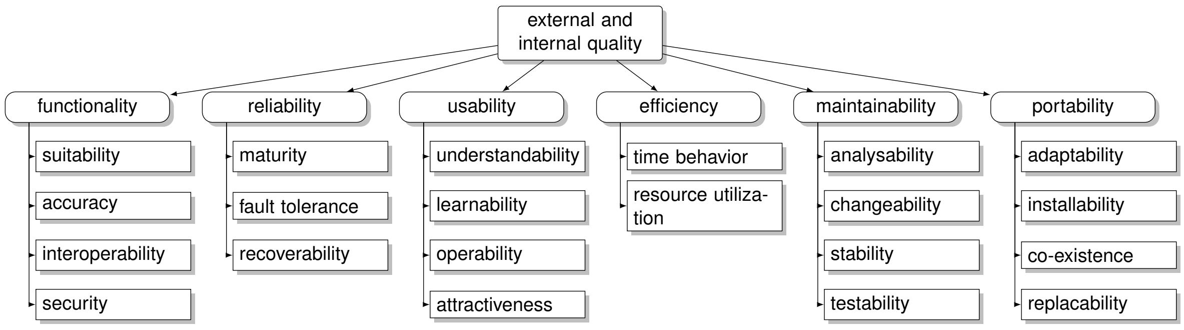


Figure 3.: The ISO 9126-1 software quality model, based on the 6 main characteristics and the corresponding sub-characteristics.

need to study further and in detail these important quality characteristics that can be understood and better individually controlled.

Performance evaluation and models

Performance is another aspect of software quality, where exists engineers currently studying the efficiency of algorithms as embodied in computational and algorithmic complexity. Response time and completion rates are some of the observable system aspects that represent performance characteristics.

Management by metrics

Using measurement as a tool for assessing the status of a project is a method more and more used by managers. The charts and graphs built according to metrics help managers to decide the best way for their projects. This becomes even more important when we have a product that does not depend entirely on software, but as a part of it has to be evaluated and thought. For certain managers that are involved in this type of business and does not have a thorough knowledge of the software is really rewarding this type of techniques that the only thing that demands is reading and understanding the data.

Structural and complexity metrics

When we want to measure some quality attributes like reliability and maintainability, some operational version of the code have to be available. When they are not, but managers still want to predict which parts of the software systems are less reliable, more testable or need more maintenance, we measure structural attributes of representation of the software which are available without the need for execution.

The (*Halstead, 1977*) and (*McCabe, 1976*) are two classic examples of defining measures that are derived from suitable representation of source code.

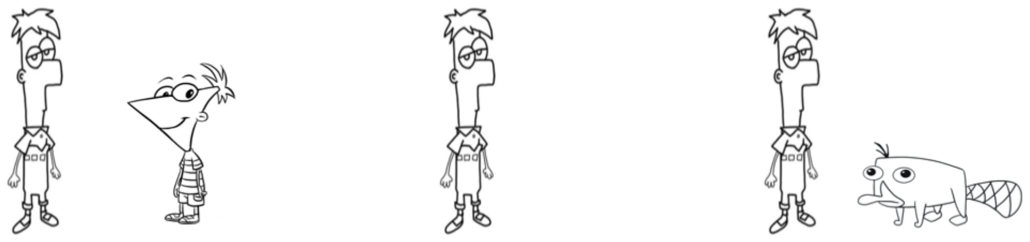


Figure 4.: We can learn about height by observing attributes in some entities. Ferb *is taller than* Phineas, Ferb *is tall* and Ferb is *much taller than* Perry.

2.1.3 *The Theory behind measurement*

As has been said, in the previous sections, we use measurement in everyday life to understand, control, predict and improve the activities that we do and the way we do it. Applying this technique in several areas, on non-software entities, we use tools and principles that are taken for granted and we do not think on the scientific basis behind it. But when we face software entities, the same tools and principles cannot be as easily applied. A deep understanding of software attributes is needed complemented with some sort of ”applied”[12] rules for measurement.

Questions about software entities are difficult to answer, such as:

1. Do we really know everything about an attribute to consider measuring it? The ”complexity” of programs is well defined for us to be able to measure it?
2. How do we know if we have really measured the attribute we wanted to measure?
3. What can we state about an attribute and the entity that posses it?
4. What meaningful operations can we perform on measures?

so we must establish the basis of a theory of measurement.

The theory that will be presented shortly is the *representational theory*[13] of Measurement according to the author[Fenton and Pfleeger (1998)]. First the concept of intuitive understanding is crucial because we need to identify and understand the attributes of a well defined entity. After that, we capture our intuitive understanding about the attribute by assigning numbers or symbols to the entities. According to the data received, we observe, and extrapolate relations between entities, that are named as **empirical relations**. Is the nature of human beings perceive things by comparing them, rather than assigning numbers or symbols to them.

12 There is no set of rules entirely well-defined and established on the art of measurement, we can only learn and take profits of the daily lessons activities.

13 Norman Fenton addresses to this theory as the theory who *”seeks to formalize our intuition about the way the world works”*.

Measurement supports a mapping between entities in the real world and values in the numeric system, for example *"Ferb's height is 13"*, but must preserve the empirical relations of the real world in the numeric system. In other words and following the previous examples, if exists a relation *"is taller then"* between Ferb and Phineas, in the real world and a mapping is done where the height of Ferb is linked to 17 and Phineas to 13, then the relation must also be valid in the numeric system, in this case, *height(Ferb) > height(Phineas)*. We call this the **representation condition**.

To achieve the desire purpose in measure, we need to interpret the data in the numeric system and draw conclusions. Different kinds of mapping can be made and this affect the type of analysis we can do, because scale types will be different for each attribute. There is a total of five scale types: nominal, ordinal, interval, ratio and absolute. According to each one, we can meaningfully make distinct kinds of statement about measures. For example, we can compute means for ratio-scales measures, but not for ordinal measures; medians can be computed for ordinal-scale measures but not for nominal-scale measures.

Most of the attributes, in software engineering, are not directly measurable, therefore we need to perform an extra effort to combine the vector elements into a larger, indirect measure.

2.1.4 *A framework for software measurement*

As mentioned in the previous subsection, there is not a concrete theory that dictates if a set of established metrics is valid or not, but there is a goal-based framework for software measurement, that can contribute to software organizations practices. The core of this framework lies on three principles: classifying the entities to be studied, determining relevant measurement goals and identifying the level of maturity reached.

In software this means that an entity can be categorized as a Process, a Product or a Resource[14], and each attribute can be classified as **internal** or **external**, depending if it can be measured only by studying the product, process or resource, or also by its behavior.

The main question, to software-engineering problems, in practice lies in which metrics should be used and how they can be applied. The common practice in software measurement is to capture product internal and external attributes, resource attributes (such as productivity and tools) and process attributes, but for the sake of this thesis, only product attributes will be explored. Just as Fenton and Pfleeger presented in [Fenton and Pfleeger (1998)], and for most of the cases in software measurement, there are two main internal attributes to measure: **size** and **structure**.

> "Internal product attributes are important, and measuring them directly allow us to assess early products and predict likely attributes of later ones."

Size is the traditional attribute to measure in software, because is useful, easy to measure without having to execute the system and because software development is besides everything, a physical

14 Processes are aggregations of software-related activities; Products are deliverables that results form software activity; Resources are entities required by a process activity.

entity. The software product size can be described in four main aspects: **length** is the physical size, **functionality**, what is really extracted by the user, **complexity**, and **reuse** size of reused product.

Relatively to the length, there are three aspects whose size is worth to know: the code[15], the specification and the design[16]. To measure functionality three approaches are used: function points, which relate the functionality with the specification; object points and specification weight. Complexity is hard to measure and its necessary to distinguish its types: problem complexity, measure the complexity of the problem; algorithmic complexity, measure the complexity of the algorithm implemented to solve the problem; structural complexity, measure the structure of the software used to implement the algorithm; and cognitive complexity, which measure the effort required to understand the software.

Another common internal attribute to measure is structure. It is proven that not always a bigger module takes longer to specify, design, code and test than a small one, the structure of the product affects its maintenance and development effort. Control-flow structure, data-flow structure and data structure are the pieces of structure. Control-flow structure reflects the interactive and looping nature of program, normally measured with direct graphs; Data-Flow structures measures the behavior of the data as it interacts with the program; Data structure measure the organization of the data itself, independent of the program.

The main goal of software engineering is to improve the quality of software products. In a measurement point of view, quality must be very well defined in terms of specific product attributes of interest of the user. From the knowledge presented to this section, anyone who desires to know how to measure the extent of his software product, is able to set targets for quality attributes. Therefore, and for the semantic flow of this document, the quality of the software product that will be studied is going to be defined and the attribute to capture that quality will be specified. A model for measuring that product will be created, starting from this framework and adapting it, to target the specified attributes. In other words, another different attributes, among the ones that were described above, will be specified in order to achieve the exact characteristics that are required to assess the product.

In a software measurement project, even if some of the metrics that are defined to be measured, for some reason, appear to be direct and at first sight usefulness, it is very important not to forget that each attribute captures a key aspect of a software quality characteristic and in the words of Fenton and Pfleeger:

> ”Those who rejects a measure because it does not provide enough information may be expecting to much of a single measure.”

15 Include the traditional measures such as lines of code(LOC), comment lines, data declarations and others

16 The specification length can be good pointer of how long the design is likely to be and the same can be said about the design length relatively to code length.

2.2 QUALITY OF LANGUAGES

As told in the previous sections, when we are trying to measure some attribute about some entity, first we need to characterize the attribute for that entity. In this master work, the attribute to be assessed is the Quality and the entity is an Attribute Grammar. But before characterizing attribute grammars we will characterize Languages, because there is a strong relationship between both concepts [17], as already said before in the Introduction section of this document.

To discuss the quality of languages in software is very important, for all kinds of reasons such as picking the work language for everyday activities, for solving complex problems, to improve the programmer quality of the work, and so on. And its important to do it in terms of **efficiency** for the recognition process of the sentences and **legibility** for easy learning, easy understanding and easy writing.

Definition 1 (Quality of a Language)**.** *The quality of a language is assessed in terms of how easy it can be learned, used and understanded, and in terms of the efficiency with which its sentences are processed.*

According to the Henriques [Henriques (2013)] personal opinion and his understanding of others ideas, such as Hoare[Hoare (1973)], Howatt[Howatt (1995)], Sebesta[Sebesta (2009)] e Watt[Watt and Findlay (2004)], it is fundamental to find a set of characteristics allowing a clear and objective reasoning about the properties that contribute for the quality of the languages and then on top of it a way to assess them. According to the author above, there are eight main characteristics that contribute to define language quality:

- Expressiveness
- Documentation
- Unicity
- Consistency
- Extensability
- Scalability
- Reliability
- Modularity

In the sequel, for each one of the features listed above, it will be provided an explanation of how that item affects each one of the four critical factors characterizing the quality of a language:

17 This section is based on the ideas and definitions proposed and discussed for Rangel Henriques in [Henriques (2013)].

- learning
- writing
- understanding
- efficient recognition

EXPRESSIVENESS can be defined as the ability to expose clearly and naturally the message to communicate, or in other words, the ease with which one can write the instructions you want to transmit. According with Pedro Rangel Henriques, exists three main elements that affect this characteristic:

- **Basic Operators** - languages that allow the used of several basic operators are more expressive, because a more natural, simple description can be made;
- **Clarity** - the used lexicon, such as the name of the operator and the signs, and the operations syntax;
- **Abstraction** - the complexity level of the operators and operations, relating with low level code.

As is expected the expressiveness characteristic affect positively the learning, writing and understanding factors because it allows saving time and effort in productivity due to the ease of language expression, but not affect the efficiency of recognition.

DOCUMENTATION is the aptitude to improve the writing of the message in some language with extra information about that message. The common example of documentation in languages are comments[18]. Some languages provide a syntax itself within these comment lines for entering data for documentation.

This characteristic affect positively the understanding factor, because more information of the message is provided, but affect negatively the writing factor because forces the sender to write more words. For the others two factors, documentation slightly affect the efficiency of recognition and does not affect the understanding.

UNICITY can be described as the capacity that a language has the form of writing operations, if there is only one way to present its instructions or if there is a choice of directions to set the same.

Clearly, unicity speeds up the learning time, the recognition time because increase its efficiency and facilitates the understanding, if there is only one way to write an instruction it is easy to the programmers to understand the sentences. In the opposite direction, unicity affect negatively the writing factor, because takes the liberty to express the ideas in the way that suits more to the person.

18 Normally this piece of information can be placed in every point and its limited by the end of the line or in case of multi-lining could involve the use of specifics symbols, to differ this information from the rest of the message.

CONSISTENCY is revealed by a language when the way of writing the same kind of ideas is consistent, always the same. Usually seen this kind of feature for its negative side, ie, many languages have, in fact, inconsistencies, either by their different syntax for similar data or be updated over time by someone who is not the original creator.

It is also important to highlight, in this characteristic, another property that plays a role in the quality of a language, just like consistency, the **orthogonality**. A language is orthogonal if the set of operators can be applied to all data types available. Consistency and orthogonality affect positively the learning, writing and understanding factor but they do not affect the recognition efficiency.

EXTENSABILITY is the ability that a language have to offer procedures to be modified or improved. There are two kinds of extensions: static extensions, which extend the language in compilation time and dynamic extensions which allow to modify a program compiled just in the run time.

The extensability speeds up the writing time because it allows user to create their own vocabulary and, syntax and semantic constructors, using them with efficiency from there on. Clearly, this characteristic have a negative effect on the learning factor and the recognition efficiency factor. As for the understanding factor, its not safe to say that have a positive or a negative effect because in one way the appearance of new constructions will difficult the understanding process but one the other side if the constructions made by the programmers are simple, could have a positive effect on understanding.

SCALABITY can be described as the language attribute where with a succession of increasingly complex problems it maintains its characteristics and quality of a uniform manner. A language is called scalable if allows to write, understand and process long sentences in the same way that allow short ones.

Scalability does not degrade the recognition efficiency and preserver readability, in other words, does not change the ease of learning, understanding and writing.

RELIABILITY is the characteristic that a language have to offer mechanisms and procedures to increase the reliability that a user have, when executing his code. This reliability is specified in terms of safety of the program and in terms of performing without errors, all the tasks for that was designed.

To increase the reliability that a user or a programmer have in a language there is some useful facets to hold, such as, a very clearly defined, unambiguous and precise semantics, one rigid type policy and a supply of constructions for error control during the execution of the program. As is easy to predict, this characteristic increase the recognition efficiency and the time of write, but facilitates the understanding factor. As for the learning factor the reliability as no considerable effect.

MODULARITY is the capacity that a language have to offer support for the writing of modules. Modules are unities or individual components that contains parts of the message and their benefits are essentially code reutilization and ease of writing.

Modularity benefits the time of writing, as it is expected, and facilitates the understanding factor, because the code is better defined and structured, parts of the code with the same purpose or operation are put together in the same module. The upside down of this characteristic is that slightly affect, in a non positive way, the learning process, because it is necessary to integrate different modules in the code, and the recognition efficiency, to the extent that there is more components to process and more data to store and manage.

SUMMARY The following table briefly summarizes all that was exposed on the influence of features chosen over the four critical factors identified above, to assess the quality of a language.

Characteristics vs Factors	Learning	Writing	Understanding	Recognition
Expressiveness	$+$	$+$	$+$	x
Documentation	x	$-$	$+$	x
Unicity	$+$	$-$	$+$	$+$
Consistency	$+$	$+$	$+$	X
Extensability	$-$	$+$	$+/-$	$-$
Scalability	$=$	$=$	$=$	$=$
Reliability	x	$+$	X	$-$
Modularity	x	$+$	$+$	x

Table 1.: The influence that Language Characteristics have on Quality Factors

(Legend) Table 1 was completed according the following criteria:

$+$ (**positive effect**) - helps to facilitate the factor in question

$-$ (**negative effect**) - helps to difficult the factor in question

$+/-$ (**variable effect**) - his contribution is not always the same, depends on other aspects

X (**have no effect**) - have no interference with the factor in question

x (**minimum effect**) - slightly helps to difficult the factor in question

$=$ (**conservative effect**) - helps to maintain the facility of the factor in question

By analyzing the previous table, Henriques[Henriques (2013)] postulated that, the quality of the language expressed in Definition 1, can be assessed by the presence of these characteristics, knowing that a language is so much better such as the number of characteristics that holds.

The challenge now exposed here is how to verify **automatically and objectively** if a given language holds some of the characteristics or not, since that they are not directly measurable indicators. More, how far the grammar quality can be automatically measured helping to assess the quality of the language which defines.

2.3 QUALITY OF GRAMMARS

After the approach taken in the previous section, to the concept of language and the quality requirements that should ideally be present in languages, an approach is made, in this section, to the central object of study in this master work - **Grammars**[Henriques (2013),Power and Malloy (2004),Klint et al. (2005),Power and Malloy (2000)] . Before submitting the necessary formal definitions for speech specification, an informal and more abstract vision of the term will be presented.

The word grammar comes from the greek meaning *grammatiké*, which is formed by the junction of the word *gramma*(weight or measure) plus the word *temática*(theme, center or focus), therefore can be classified as ”focus control”. In a more abstract view, a grammar can be seen as a skeleton of the language, in the sense that as a skeleton defines an objective structure, shape or composition, in the human body, a grammar defines a language.

Linguistically, a grammar is a set of individual rules used to regulate the language and to establish writing patterns, presenting unities and structures that allow good use of language through the validation of sentences from an alphabet according to the syntax. The art of putting the right words in the right places.

Now bringing these terms to the field of computer science, it is inevitable not to mention Noam Chomsky[19], for his famous work in the field of formal languages, where he proposed a hierarchy to classify formal grammars into groups/classes - *Chomsky Hierarchy*[20]. Typically, in software systems, grammars are used to describe formal languages such as programming languages, where the syntax of the language is specified using the Chomsky grammatical model known as Context-Free Grammars(CFG) [Knuth (1968)], which will be defined below. To give semantic meaning to sentences and restrict some senseless syntactic constructions, typically are used Attribute Grammars (AG) [Deransart et al. (1988)], which will be also defined in the next subsection.

The languages are structurally described by grammars, and any language can be defined by a number of different grammars. That why, it is desirable, that the quality of the grammar affect the quality of the language generated.

2.3.1 *Formal Grammar Definitions*

Definition 2 (Context-Free Grammar)**.** *A Context-Free Grammar (CFG) is defined by the four-tuple:*

$$CFG =< T, N, S, P >$$

19 Linguistic, philosopher, cognitive scientist and still a Linguistic Professor in MIT(Massachussets Institute of Technology), is known as the ”father of modern linguistics” and responsible for a major contribute to the field of automatic process languages. He is the author of vast literary collection of books and articles which first appeared the term of generative grammar. For further more information about this author, please check http://www.chomsky.info/.

20 This hierarchy is composed by four levels: Type-0 unrestricted grammars, which includes all formal grammars and can generate arbitrary recursively enumerable languages; Type-1 context-sensitive grammars which generate context-sensitive grammars; Type-2 context-free grammars which generates context-free languages, recognized by a non-deterministic automaton; and Type-4 regular grammars which generates regular languages, recognized by a finite state automaton.

where T is the (finite) vocabulary of the ***terminals symbols*** *of the language, N represent the* ***non-terminals symbols*** *set of the grammar, S is the* ***start symbol*** *of the grammar where $S \in N$, and finally P is the* ***set of productions*** *or derivation rules of the grammar. The T set of terminals symbols is divided in 3 disjoints subsets - $T = RW \cup Sig \cup TV$ - the* ***Reserved-Words****, the* ***Signs*** *and the* ***Terminals-Variables****.*

A grammar defines a language by specifying valid sequences of derivation steps which comprises to a sequence of symbols(sentences of the language). The exercise performed to derive a phrase from a grammar is: from the start symbol S and applying the production rules, replacing the non-terminals for the right side of the production until only terminal remain.

Each production $p \in P$ is a rule formed by:

$$p : X_0 \rightarrow X_1 \ldots X_i \ldots X_n$$

in which p is the rule identifier, $\rightarrow$ is the derivation operator, $X_0 \in N$ and $X_i \in (N \cup T)$ with $1 \leq i \leq n$. The production p should be read - from the left side of the operator, also designated LHS(p) which is always a non-terminal symbol, to the right side of the operator RHS(p), a sequence of non-terminal and terminal symbols - as "non-terminal X_0 derives phrase $X_1 \ldots X_i \ldots X_n$".

Now, that a production rule is defined, it is important to characterize a unit production, once this will be relevant in the following subsections.

Definition 3 (Unit Production)**.** *A unit production is a production $up \in P$ with only one symbol in the right side ($\#RHS(p) = 1$), such as:*

$$X_0 \rightarrow X_1$$

noting always that $X_0 \neq S$

Following the definitions, it is presented below a small CFG example, serving as a simple case of study, which define a language called **List**.

Example 1. *This grammar say that a sentence in `List` is a `Content` surrounded by brackets and a Content is either an `Item` or either an `Item` followed by a comma and a `Content`. An `Item` is a atomic value, number (**`num`**) or a word (**`wrd`**).*

```
T = {num, wrd, '[', ']', ','}
N = {List, Content, Item}
S = List
P = {
       p0: List    -> '[' Content ']'
       p1: Content -> Item
       p2: Content -> Item , Content
       p3: Item    -> num
```

```
    p4: Item     -> wrd
}
```

According to the example, a list must always have elements of any kind and in any order. A valid sentence of the generated language by this grammar is:

$$[3, sad, tigers]$$

and an invalid sentence is, for example:

$$[, sad, 1, tigers]$$

One way to verify that a CFG is well written is trough the notion of well-formed grammar:

Definition 4 (CFG well-formed)**.** *It is said that a Context-Free Grammar is well-formed if:*

- *for all $X \in N$ exists at least 1 production with X on the left;*
- *for all $X \in N$, X is* ***reachable****, this is, exists at least 1 derivation form the axiom that uses X;*
- *for all $X \in N$, X is* ***terminable****.*

Complementing the previous definition,

Definition 5 (Terminable Symbol)**.** *A symbol it is called terminable if:*

- *is a terminal symbol;*
- *is a non-terminal symbol and exists at least 1 production with this symbol on the left which his right side is a terminable sequence.*

wherein a sequence of symbols $N \cup T$ ***is terminable*** *if:*

- *is the empty sequence;*
- *each symbol of this sequence is terminable.*

To clarify the terms introduced before, an example is presented above.

Example 2. *Despite being uncommon, the grammar of* ***List*** *language, is well-formed because:*

- *exists at least 1 production for each one of the 3 non-terminals symbols:* `p0` *to* `List`*;* `p1, p2` *to* `Content`*;* `p3, p4` *to* `Item`*.*
- *the two non-terminal besides the axiom are reachable from* `List` *:* `Content` *is used directly in* `p0`*; and* `Item` *is used indirectly (via* `Content`*) in* `p1`*.*

- *all non-terminals are terminables:* `Item` *is terminable thanks to* `p3` *because his RHS is a terminal(or* `p4`*, for the same reason);* `Content` *is terminable because* `p1` *in which the RHS is a terminable sequence;* `List` *is terminable because in the $RHS(p0)$ all symbols are terminables.*

Another set of relevant definitions for grammars and for better characterization, both in practical terms of implementing the processors or in visually terms of derivation rules flow, is the concepts of: Derivation Tree or also as known as Parsing Tree, and Dependency Graph, both will be defined above.

Definition 6 (Derivation Tree of a production)**.** *Each production $p \in P$ have a **derivation tree** which the root is X_0 and the descendants are the n symbols X_i from the right side.*

Example 3. *In the case of the production rule $p0 \in P$ defined in the Example1:p0 : ' [' Content '] '*

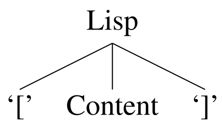


Figure 5.: The derivation tree from production rule $p0$, where we can see the root as the left side symbol of $p0$ and the three descendants from the right side as leaves.

Definition 7 (Derivation Tree)**.** *The derivation tree of a grammar is a tree whose root is the start symbol, whose nodes are non-terminals and whose leaves are terminals. The children of any node in the tree correspond to those symbols on the right hand side on a production rule. Taking the previous definition, the tree is the junction of each derivation tree from the productions.*

Definition 8 (Dependency Graph between symbols)**.** *It is given the name of **Dependency Graph between Symbols** (DGS), to a graph in which his vertices are symbols $N \cup T$ from the grammar and the branches or arches, go from the Y vertex to the X vertex every time that exists in P one production p such as $p : X \rightarrow \ldots Y \ldots$, concluding then that X **depends on** Y or that Y **derived from** X.*

Example 4. *In the case of the* List *grammar, specified in Example 1, the DGS is illustrated on Figure 6.*

As pointed out before, to specify the semantics of the language is used a Attribute Grammar, defined as:

Definition 9 (Attribute Grammar)**.** *A Attribute Grammar (AG) is defined by the tuple:*

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

where,

$A = \bigcup A(X), \forall X \in (N \cup T)$ *is the set of **attributes** of all symbols of the grammar.*

$CR = \bigcup CR(p), \forall p \in P$ *is the set of attributes **calculation rules** in all productions of the grammar.*

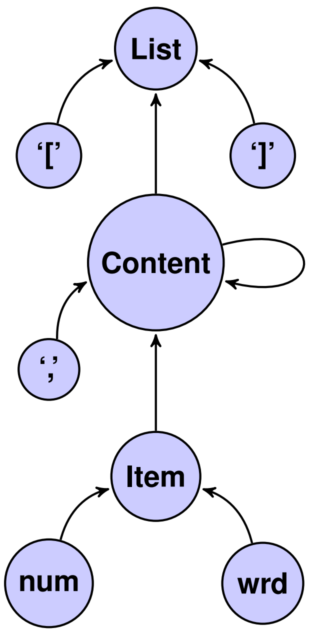


Figure 6.: The DGS of the *List* grammar, where are included the branches of the grammar.

$CC = \bigcup CC(p), \forall p \in P$ is the set of ***context conditions*** *in all productions of the grammar.*
$TR = \bigcup TR(p), \forall p \in P$ is the set of ***translation rules*** *in all productions of the grammar.*
The attributes $A(X)$ of each symbol are devised in two distinct finite subsets

$$A(X) = Inh(X) \cup Syn(X), Inh(X) \cap Syn(X) = \emptyset$$

in which,
$Inh(X)$ is the ***inherited attributes*** *set of the symbol X, the ones that carry the information down through the sub-tree.*
$Syn(X)$ is the ***synthesized attributes*** *set of the symbol X, the ones that synthesize the information from the leaves and carry it up through the tree.*

For each instance of the symbol X in the derivation tree, $X \in N \cup T$, it will be saved a set of properties (the concrete values of his attributes), characterizing from the semantic point of view. The term of **decorated derivation tree** arises in this context, where the derivation tree in filled with the properties of each symbol, in other words, carrying the meaning of each node.

The calculation rules indicates how valuing the attributes, to obtain the meaning. With the meaning of each symbol, the meaning of the sentence is built and so, it is possible to transform, or translate, the sentence to obtain the desired result. However, for the sentence to be processed two things have to happen, the CFG productions have to ensure syntactic correction; and the AG context conditions have to ensure semantic validation, explaining the restrictions which the concrete values of attributes have to hold for a sentence to make sense.

Definition 10 (AG well-formed)**.** *It is said that a Attribute Grammar is well-formed if:*

- *the underlying CFG is well-formed;*
- *for each production $p \in P$ is provided one and only one rule of the form*

$$X_i.a = f(\dots X_j.b \dots);$$

  to the attributes evaluation a from X_i, may be used the attribute b from X_j, according with the following conditions:

  - *The attributes a to evaluate have mandatorily to be the synthesized from X_0 and the inherited from X_i, $1 \leq i \leq n$;*
  - *In the calculation formula can be used the attributes b inherited from X_0 or synthesized from X_j,$1 \leq j \leq n$.*
- *the induced Dependency Graph upon each derivation tree by that calculation rules is acyclic.*

2.3.2 *Assessing Grammar Quality*

Following the speech of reasoning so far presented and to integrate the scientific method presented earlier in this chapter, the need arises, to assess the quality of a grammar. Similarly to the characterization of Languages, exposed in this document, it is intended to characterize also the grammars, with the ultimate goal compile a set of software metrics that will quantitatively assess the quality criteria then presented.

According to the train of thought presented in the Grammatical lesson performed by Pedro Rangel Henriques, a grammar is responsible for two activities:

- to define or generate the language, and therefore validating the sentences according to the syntax;
- to guide the recognition of the language phrases that generates, being this function crucial because it allows to derive automatically and systematically the programs responsible for the process of the sentences (recognition and transformation).

The characteristics that allow to assess grammar quality and comparing them are:

- **Usability** *while language generator* of the grammar as tool for sentences derivation of a language:
  - ease of understanding
  - ease of derivation
  - ease of maintenance
- **Efficiency** *while program generator* of the grammar as tool for language processors derivation:

– efficient recognition of sentences from the generated language

– efficient processor automatic generation

Definition 11 (Grammar Quality)**.** *The quality of a grammar,* ***while specification that generates a language****, is recognized if it facilitates its usability, in other words, if it easy to learn it (understanding what describes), to use it for derive sentences and to maintain.*

The quality of a grammar, ***while specification that generates a processor****, is assessed by the program efficiency, which from it derives, and the efficiency of the own generation process. Then, it is assumed that a grammar hols quality if allows the generation of efficient language processors without degrade the ease of automatic generation.*

As the definition of Attribute Grammar evolved through the definition of Context-Free Grammar, first will be debated the characterization of Context-Free Grammars, in which the syntax of the language is defined and then Attribute Grammars, which specify even the language semantic.

Context-Free Grammars Characterization

USABILITY can be referred as the clarity and ease with which the entities interact with grammars. It is important distinguish two groups of entities that handle grammars in different perspectives: the final user, that seizes the grammar as a tool for the transmission of a message, in the linguistic aspect, in which case the usability is assessed by ease with which he read and use the grammar , while instrument for deriving sentences of the language; the grammar engineer, that seizes the grammar in a technical aspect, concerned that the grammar fulfills efficiently the functions for which it was designed, in this case the usability is assessed by ease with with he understand and maintain the grammar.

The **understanding** criteria is related to:

- choosing the identifiers for the non-terminals and terminals symbols
- the use of unit productions
- the length of the right sides of productions
- the notation employed
- the type of recursion (right or left, direct or indirect).

Looking now for **derivation** criteria, reducing the number of productions and the number of non-terminals facilitates the process, such maintaining of the same notation and using clear identifiers.

Finally, for the **maintenance** criteria, besides all that was exposed before, two more elements emerged as important: modularity, in the process of creating and maintaining a grammar with the reuse option, importing pieces of another grammars, and complexity, in the way how symbols depend on each other.

EFFICIENCY **in program generation** can be seen as the way that the grammar writing affect the generated processor efficiency and the own generation process.

The **Efficiency Recognition** of the sentences from the generated language is measured in terms of:

- *Parsing* time
- size/complexity of the *Parsing Tables*

The increasing of the number of symbols and productions implies an increase of *Parsing Tables* size, or control structures. This event does not affect the recognition time, but in practice slightly degrades it. The increasing of the right sides of productions will lead to an increase of the used memory, in the recognition operation, but in consequence will decrease the time, due to less parsing operations.

As for the **Efficiency in automatic Generation of processor** is a characteristic that depends on:

- generation time
- the data structures size, used for storing and transforming the grammar

An increase on the grammar size (number of symbols and productions) implies an increase in the generation time and in the storage space for that process. The complexity and the recursion system does not affect the efficiency in this generation phase, but on the other hand, the modularity degrades the processing time of the grammar, because have more modules to open and analyze.

SUMMARY The following table briefly summarizes all that was exposed on the influence of features chosen in the four critical factors identified above, to assess the quality of a Context-Free Grammar.

Elements vs Factors	Usability			Efficiency	
	Understanding	Derivation	Maintenance	Recognition	Aut.Generation
Clear Ids Symbols	+	+	+	*X*	+Ti,Si
Unit Productions	+	−	+	+Ti,Si	+Ti,Si
RHS Length	−	+	−	*x*+Ti,Si	*X*
Notation	+/−	−p, +ex	−p, +/−ex	*X*	*X*
Recursion System	+/−	+r, −l	+/−	−Ti,Si	*X*
Modularity	−	−	+	*X*	+Ti
Syntax Complexity	*X*	*X*	−	*X*	*X*

Table 2.: The influence that CFG elements have on Quality Grammar factors.

(Legend) Table 2 was completed according the following criteria:

+ (**positive effect**) - helps to facilitate the factor in question

$-$ (**negative effect**) - helps to difficult the factor in question

$+/-$ (**variable effect**) - his contribution is not always the same, depends on other aspects

X (**have no effect**) - have no interference with the factor in question

x (**minimum effect**) - slightly helps to difficult the factor in question

Ti - Processing/Generation Time

Si - Size of the Data Structures for supporting the processing/generation

p - pure-BNF

ex - ex-BNF

r - right recursion

l - left recursion

Attribute Grammars Characterization

Under the premise that an Attribute Grammar has always themselves an underlying Context-Free Grammar, it is expected that the characterization should be made completing the characterization of a CFG with the study of semantic behavior, implied by the AG, in terms of usability and efficiency. To accomplish that, the same procedure will be followed, as it was for the previous subsection, where the general characteristic of usability is dismantled in three components: ease of understanding, ease of derivation and ease of maintenance, and the general characteristic of efficiency is dismantled in two: efficiency in processing and efficiency in automatic generation of processor.

USABILITY The **understanding** criteria of an AG, besides all the elements appointed for CFG, also makes sense to add the following specific elements of this type of grammars:

- the right choice for the attributes identifiers
- the attributes complexity
- the number of attributes
- the number of calculation rules, context conditions and translation rules
- the notation used and the simplicity in which attributive operations are written

As is easily perceptible, the choice for clear and concise identifiers for attributes, describing well what they represent, affects positively the understanding of the grammar, such as the lower complexity of the attributes, in a way that, attributes that store structured values are more difficult to understand then attributes storing atomic types.

For knowing how the number of attributes and operations affect the understanding is not so noticeable. As might initially think, as the number of both increase the difficulty in understanding increases in a proportional way, however, if the number of attributes grown but his type is simpler, as well as less elaborate operations, the ease of understanding could actually decrease.

Relatively to the notation or the programming languages in which the attributive operations are written, declarative languages have advantage to the understanding of the calculation rules, context condition and translation rules, because they handle complex data type more easily then imperative languages.

The recursion system element does not affect to much the understanding, the pure-BNF notation allows a more systematic, simpler and clear writing, the modularity although allow to maintain the grammar compact and organized, in this case end up to difficult the legibility and finally the complexity between symbols and attributes have no significant influence on the understanding criteria.

In case of **derivation**, there is little to add, the elements mentioned in the characterization of CFG are still valid and can only mention that for the derivation only the context conditions give some information to the user of the language. The right location of the context conditions on productions and the way that they are clearly written helps the user on the derivation process, for valid sentences.

To facilitates the grammar **maintenance**, intervenes again all identified elements for CFG, emerging only an important factor: the semantic complexity. The way that attributes depend on each other- how many other attributes an attribute need to be calculated- affect the maintenance, the simpler the easier it is to keep it.

EFFICIENCY The **Efficiency in Processing** the sentences of the generated language is a characteristic that is measured in terms of:

- time of analysis and translation
- size/complexity of the structures that guide the analysis and translation

On this topic, it must be stated that in terms of the AG size, the number of attributive operations affect the verification and translation effort, increasing the time according to that number. But a AG can have a higher number of attributive operations in relation to another, and still performing them better, if they have less complexity. The number of symbol, productions or even attributes, will lead to a bigger internal structure for verification/translation support and so, besides increasing the memory consumption also increase the crossing time on the abstract syntax tree, which implies an indirect conditioning on the time of processing.

The **Efficiency in automatic generation of processor** is a characteristic that is measured in terms of:

- generation time
- size of the internal data structures used for grammar storage and transformation

What can be said, in objective and generic terms, in this case is that an increasing of the AG size (number of symbols, number of productions, number of attributes and attributive operations) implies also an increasing on the generation time and on the storage space required during that process. The difference for the syntax complexity discussed for CFG, is that now the semantic complexity strongly affect the generation process due to the determination of the total order to performing attributive operations during the decorated abstract syntax tree traveling. Therefore, the generator efficiency decreases when the semantic complexity rises.

SUMMARY The following table briefly summarizes all that was exposed on the influence of features chosen in the four critical factors identified above, to assess the quality of a Attribute Grammar.

Elements vs Factors	Usability			Efficiency	
	Understanding	Derivation	Maintenance	Recognition	Aut.Generation
Clear Ids Attributes	$+$	X	$+$	X	$+$Ti,Si
N^o Attributes	$\surd$	X	$-$	xTi,Si	$+$Ti,Si
N^o Attr.Operators	$\surd$	X	$-$	$+$Ti	$+$Ti,Si
N^o Symbols$+$Prod.	$\surd$	$-$	$-$	xTi,Si	$+$Ti,Si
Attr.Complexity	$-$	x	$-$	$+$Ti,Si	X
Attr.Opert.Complexity	$-$	x	$-$	$+$Ti,Si	X
CCs Placement/Clarity	$+$	$+$	$+$	X	X
Notation	$+$p, $-$ex	x	$+$p, $-$ex	X	X
Recursion System	$+/-$	$+/-$	$+/-$	$-$Ti,Si	X
Modularity	$-$	$-$	$+$	X	$+$Ti
Semantic Complexity	X	X	$-$	X	$+$Ti,Si

Table 3.: The influence that AG elements have on Quality Grammar factors.

(Legend) Table 3 was completed according the following criteria:

$+$ (**positive effect**) - helps to facilitate the factor in question

$-$ (**negative effect**) - helps to difficult the factor in question

$+/-$ (**variable effect**) - his contribution is not always the same, depends on other aspects

$\surd$ (**dependent effect**) - has influence, but the signal depends on other aspects

X (**have no effect**) - have no interference with the factor in question

x (**minimum effect**) - slightly helps to difficult the factor in question

Ti - Processing/Generation Time

Si - Size of the Data Structures for supporting the processing/generation

p - pure-BNF

ex - ex-BNF

r - right recursion

l - left recursion

2.3.3 *Metrics for Grammars*

Metrics for Context-Free Grammars

Assuming G as a *well-formed* Context-Free Grammar and DGS as the respective Dependency Graph between symbols, this document exposes the metrics to assess the quality of G, dividing them into 3 groups:

- Size Metrics:
  - (SM1) **Grammar size**, measured in terms of:

Parameter	Description
#T	number of terminal symbols
#N	number of non-terminal symbols
#P	number of productions
#UP	number of unit productions
#R	number of symbols directly or indirectly recursive
§RHS	average number of symbols in the right hand sides
§RHS-Max	maximum number of symbols on a right side, $max(length(RHS))$
§Alt$^{(1)}$	average number of alternative productions for the same left sides
§Alt-Max	maximum number of alternative productions for the same left side
#Mod	number of imported grammatical modules

Table 4.: Metrics for assessing the Size of Context-Free Grammars.

  - (SM2) **Grammar syntax complexity**, measured in terms of:

Parameter	Description
FanIn$^{(2)}$	average number of branches of the input nodes (non-terminals) of the DGS
FanOut$^{(3)}$	average number of branches of the output nodes of the DGS

Table 5.: Metrics for assessing the Syntax Complexity of Context-Free Grammars.

  - (SM3) **Parser size**, measured in terms of:

Parameter	Description
#RD	number of functions from the Recursive-Descent Pure Parser $(\#(N \cup T))$
§TabLL	dimension of the Parsing Table LL(1) $(\#N \times (\#T + 1))$
§$\mathcal{DA}$-LR	dimension of the Deterministic Automaton LR(0) $(\#Q)$
§TabsLR	dimension of the Parsing Tables LR(0) $(\#Q \times (\#T + 1); \#Q \times \#N)$

Table 6.: Metrics for assessing the Parser Size of Context-Free Grammars.

Notes:

1. To the calculation of §Alt, it is assumed that each non-terminal symbol has always 1 alternative; therefore, this value is $\#P/\#N$.

2. **derivation factor**, measure how many symbols derives from one symbol.
3. **dependency factor**, measure the times that one symbol is used in other symbols definitions.

- Style Metrics
  - (FM1) **form of Recursion**, may have one of the following values:

Value	Description
DirectRec	all recursion cases follow the pattern $X \rightarrow \ldots X \ldots$
IndirectRec	all recursion cases follow the pattern $X \rightarrow \ldots Y \ldots ; Y \rightarrow \ldots X \ldots$
FMixedRec	both forms of recursion are used

Table 7.: Metric for assessing the form of Recursion of Context-Free Grammars.

  - (FM2) **type of Recursion**, may have one of the following values:

Value	Description
RecR	DirectRec cases follow the recursive right pattern $X \rightarrow \epsilon \mid e\ X$
RecR-LL	DirectRec cases follow the recursive LL(1) pattern $X \rightarrow e\ C ; C \rightarrow \epsilon \mid e\ X$
RecL	DirectRec cases follow the recursive left pattern $X \rightarrow \epsilon \mid X\ e$
TMixedRec	several direct recursion patterns are used

Table 8.: Metric for assessing the type of Recursion of Context-Free Grammars.

  - (FM3) **notation**, may have one of the following values:

Value	Description
BNF	all notations are written in Backus-Naur Form pure
ex-BNF	all notations are written in Extended Backus-Naur Form
MixedN	both notations are used

Table 9.: Metric for assessing the Notation used in Context-Free Grammars.

- Lexicographical Metrics
  - (LM1) **clear identifiers** for terminals and non-terminals symbols, is calculated by the formula:

$$\#IdCompl / (\#IdCompl + \#IdAbrev)$$

considering the definition 12, the name of the concept for each symbol N or TV from the grammar and being:

Value	Description
#IdCompl	number of symbols in which the identifier *derives* from the concept name
#IdAbrev	number of symbols in which the identifier does *not derives* from the concept name

Table 10.: Metric for verifying Clear Identifiers for Non-Terminals and Terminals-Variables declared in Context-Free Grammars.

– (LM2) **reserved-words and clear signs** from the language defined by G is calculated through the formula:

$$\#RWCompl/(\#RWCompl + \#RWAbrev)$$

considering the definition 12, the name of the each concept from the language and being:

Value	Description
#RWComp	number of cases in which the reserved-word *derives* from the concept name
#RWAbrev	number of cases in which the reserved-word *not derives* from the concept name

Table 11.: Metric for assessing Reserved-words and Clear Signs declared in CFGs.

– (LM3) **flexibility of terminal-variables**, is calculated through the formula:

$$\#TFlex/(\#TFlex + \#TRigid)$$

being:

Value	Description
#TFlex	number of terminals with flexibility on the value or identifier construction (T-variable)
#TRigid	number of terminals without flexibility on the value or identifier construction (T-variable)

Table 12.: Metric for measuring Terminal-Variables flexibility in CFGs.

– (LM4) **kind of comment**, is calculated through the formula:

$$inline + block + metaI$$

being:

Value	Description
inline	1 if accepts comments from one point to the **eol**; otherwise 0
block	1 if accepts comments formed by blocks with one or more lines; otherwise 0
metaI	1 if accepts meta-information inside the comments blocks; otherwise 0

Table 13.: Metric for checking the amount of comment types in CFGs.

In order to complete the lexicographic metrics it is necessary the following definition:

Definition 12 (Identifier Derivation)**.** *It is said that a* Identifier ***derives*** *from the Concept Name if:*

1. *the identifier is equal the name;*
2. *the identifier is a prefix of the name, with 3 or more letters;*
3. *the identifier has a prefix which is prefix of the name and the other letters can be obtained from the name by removing some of them.*

The previous metrics here exposed affect differently the several characteristics, explored in the former section, to assess the grammars, as well as, by implication the characteristics to assess the languages.

Metric vs Factors	Understanding	Derivation	Maintenance	Recognition	Aut.Generation	LQ
SM1	√	√	√	√	√	√
SM2	√		√			
SM3				√	√	√
FM1	√	√	√			√
FM2	√	√		√		√
FM3	√	√	√			√
LM1	√	√	√		√	√
LM2				√		√
LM3				√		√
LM4				√		√

Table 14.: The influence between the metrics and the quality factors in CFGs.

Metrics for Attribute Grammars

Assuming AG as a *well-formed* Attribute Grammar, LDG as the respective Local Dependencies Graph for attributes and keeping in mind all metrics introduced before for the assessment of the underlying Context-Free Grammar, a set of attributive parameters for measure its quality are presented:

- Size Metrics:
  - (ASM1) **Attribute Grammar size**, measured in terms of:

Parameter	Description
#A	number of attributes
#IA	number of inherited attributes
#SA	number of synthesized attributes
#CR	number of calculation rules
#CC	number of context conditions
#TR	number of translation rules

Table 15.: Metrics for assessing the Size of Attribute Grammars.

- (ASM2) **Grammar semantic complexity**, measured in terms of:

Parameter	Description
FanIn$^{(1,2)}$	average number of branches of the input attributes of the productions LDGs
FanOut$^{(1,3)}$	average number of branches of the output attributes of the productions LDGs

Table 16.: Metrics for assessing the Semantic Complexity of Attribute Grammars.

Notes:

1. calculation is done by analyzing the Local Dependency Graph, LDG, associated with each production and taking for each attribute the maximum.
2. measure how many attributes does one attribute need for his definition.
3. measure the times that one attribute is used in other attributes definitions.

• Style Metrics

- (AFM1) **attributes complexity**, is calculated by the formula:

$$\#AComplex/(\#AComplex + \#AAtom)$$

being:

Parameter	Description
#AAtom	number of attributes pf type atomic
#AComplex	number of attributes of type structured (with/without pointers, or hashing)

Table 17.: Metric for assessing Attributes Complexity in AGs.

- (AFM2) **complexity of the attributive operations**, is calculated by the formula:

$$\#OComplex/(\#OComplex + \#OSimple)$$

being:

Parameter	Description
#OSimple	number of CR, CC, or TR formed only by 1 instruction
#OComplex	number of CR, CC, or TR formed by 1 block of instructions

Table 18.: Metric for assessing the Complexity of Attributive Operations in AGs.

– (AFM3) **calculation scheme** for writing CRs, measured by the addition of two values, one that assess the from of aggregation according to the following table:

Value	Description
CRAggreg	the attributes calculation follow the values aggregation pattern
CRNAggreg	the attributes calculation follow the values non-aggregation pattern

Table 19.: Metric for assessing the Calculation Scheme, regarding the values aggregation form, in AG.

and another that classifies the form of recursive values accumulation, according to the following table:

Value	Description
CRpureS	the calculation of the accumulation attributes follow the purely synthesized pattern
CRpureI	the calculation of the accumulation attributes follow the purely inherited pattern
CRmixIS	the calculation of the accumulation attributes follow the mixed i/s pattern
CRvar	the attributes calculation does not follow a typical pattern

Table 20.: Metric for assessing the Calculation Scheme, regarding the values accumulation pattern, in AG.

– (AFM4) **semantic restriction scheme** for writing CCs, can take the following values:

Value	Description
CCTop	the CCs collocation follow mostly the synthesized pattern
CCCentered	the CCs collocation follow mostly the right point pattern
CCBottom	the CCs collocation follow mostly the inherited pattern
CCvar	the CCs collocation does not follow a typical pattern

Table 21.: Metric for assessing the Semantic Restriction Scheme in AG.

– (AFM5) **translation scheme** for writing TRs, can take the following values:

Value	Description
TRTop	the TRs collocation follow mostly the synthesized pattern
TRInterm	the TRs collocation follow mostly the right point pattern
TRBottom	the TRs collocation follow mostly the inherited pattern
TRvar	the TRs collocation does not follow a typical pattern

Table 22.: Metric for assessing the Translation Scheme in AG.

– (AFM6) **style of the language** to the writing of the attributive operations, if it is a *declarative language*, or not (if it is imperative).

  - (AFM7) **language specificity** to the writing of the attributive operations, if it is a standard programming language or not.

- Lexicographical Metrics
  - (ALM1) **clear identifiers** for attributes, is calculated by the formula:

$$\#IdACompl/(\#IdACompl + \#IdAAbrev)$$

considering the definition 12, the name of the concept denoted for each attribute from the grammar and being:

Value	Description
#IdACompl	number of attributes in which the identifier *derives* from the concept name
#IdAAbrev	number of attributes in which the identifier does *not derives* from the concept name

Table 23.: Metric for assessing Clear Identifiers for Attributes in AG.

  - (ALM2) **clear identifiers** for attributive operators (function names, predicates and called procedures in the CR, CC and TR), is calculated by the formula:

$$\#IdOCompl/(\#IdOCompl + \#IdOAbrev)$$

considering the definition 12, the name of each operation which intended to preform in each rule and being:

Value	Description
#IdOCompl	number of operations in which the identifier *derives* from the operation name
#IdOAbrev	number of operations in which the identifier does *not derives* from the operation name

Table 24.: Metric for assessing Clear Identifiers for Attributive Operators in AG.

Following the procedure done for the Context-Free Grammars, Table 25 presents the influence of the several metrics involving attributes upon the characteristics defined before for assessing the quality of the grammar, showing also the relation with the quality of the generated language by the AG.

Metric vs Factor	Understanding	Derivation	Maintenance	Recognition	Aut.Generation	LQ
ASM1	√	√	√	√	√	√
ASM2	√		√	√	√	√
AFM1	√	√	√	√		√
AFM2	√	√	√			√
AFM3	√	√	√	√	√	√
AFM4	√	√	√	√	√	√
AFM5	√		√	√	√	√
AFM6	√	√	√	√		√
AFM7	√	√	√	√	√	√
ALM1	√	√	√		√	√
ALM2	√	√	√		√	√

Table 25.: The influence that metrics have on Quality factor of AGs.

3

THE PROBLEM AND ITS CHALLENGES

In this chapter, the characterization and delimitation of the **problem**, that this masters work aims to fulfill, is presented and discussed. After Chapter 1 contextualize the theme, and after presenting the *State of the Art*, where the theoretical foundations (such as the basis of Measurement) and the formalization of all the basic concepts were seen, arises the *Problem and its challenges* chapter.

It is necessary to define the extent to which this proposal intents to use a scientific method - Measurement - applying it to the object of study - Attribute Grammars - creating this way an Hypothesis, upon which something will be developed for the sake of assessing the quality of a grammar and, by direct consequence, the quality of the language generated by such grammar.

The problem that this masters work aims to address is, in more objective terms, the Quality of Attribute Grammars, which can be seen as an intention to overcome the flaws of the grammar-based software, an agenda advocated by many authors, just as was exposed in the Motivation section. Grammarware is present not only in programming language compilers, but also in tools for reverse engineering, program analysis, software metrics, documentation generation and detection of program anomalies. Ensuring their completeness and correctness is vital to their use, by making robust and reliable large grammars.

The objective is to develop a software application that implements the notions of grammars and language quality, performing an automatic metrics calculation, as previously defined. It is desired as final goal, an effective computable technique to assess quality of grammarware and to steer the improvement of quality. In simple words, the wish is to automate, by developing a tool, the process of assessing grammars quality, as explained before.

Although exists some tools similar to the described, such as the *SynQ* tools by Power and Malloy [Power and Malloy (2004)] and the *gMetrics* tools [Crepinsek et al. (2010)], the proposed system is something different. Despite using similar procedures, it will show different results because different metrics will be evaluated and involved in the assessment.

3.1 TOOL DESCRIPTION

Assuming G as a grammar and ML as the meta-language in which G is specified, being MG the meta-grammar that generates ML and MG an attribute grammar, the QG - Quality Grammar - system should:

- allow to define the metrics to calculate, associating synthesized attributes to the MG symbols;
- allow to define evaluation rules for those attributes, according to the respective metric meaning;
- accept a given G grammar, written in ML, and assess its quality evaluating the attributes;
- allow to manipulate G, for example, transforming it into an equivalent grammar with higher quality.

To accomplish the desired features listed above, the tool should be developed accordingly some requirements:

- accept as input two different type of grammars, defined in the previous chapter, Context-Free Grammars and Attribute Grammar;
- validate the syntax of those grammars, because for now it is intended to accept only the ANTLR[1] format;
- use a grammar that specifies the ANTLR meta-language and therefore, using the own ANTLR tool, generate the Parser and the Lexer;
- calculate automatically each metric by extracting information in the recognition process (add semantic to the grammar);
- from the evaluated metrics, perform a quality report for the input grammar and the respective generated language.

Figure 7 represents, trough an *Activity Diagram*, what was said, or in other words, the activity flow that the system should obey.

1 ***AN****other* ***T****ool for* ***L****anguage* ***Recognition*** is a parser generator that takes as input a grammar that specifies a language and generates as output source code for a recognizer for that language. A language is specified using a Context-Free Grammar which is expressed using Extended Backus–Naur Form (EBNF) notation.

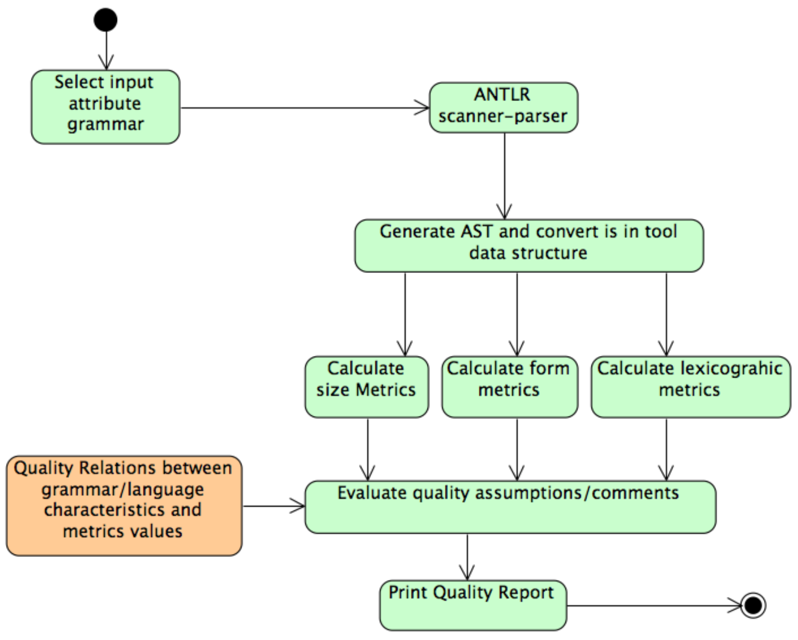

Select input attribute grammar
ANTLR scanner-parser
Generate AST and convert is in tool data structure
Calculate size Metrics
Calculate form metrics
Calculate lexicograhic metrics
Quality Relations between grammar/language characteristics and metrics values
Evaluate quality assumptions/comments
Print Quality Report

Figure 7.: Activity Diagram

4

GQE - GRAMMAR QUALITY EVALUATOR

This chapter presents itself as the main result of this thesis, the core of this document and the proof for the scientific evidence provided in the previous chapters. The main goal of this *Contribution* chapter, after in the previous chapters having responded to the questions: the *What?* in chapter 1, the *What for?* in the chapter 2 and the *Why?* in the chapter 3, is to answer now, to the *How?* question.

Following the thread of this dissertation, a new software tool for metric evaluation will be introduced, representing the obtained result with a simple purpose, evaluating a new set of metrics for context-free grammars (CFG) and attribute grammars (AG) in order to help the assessment of grammars quality. The premise is to explain completely how this tool was developed, in what architecture is settle, how it works, what are its functions and how the metrics were implemented (algorithms and structures).

Therefore, the first section of this chapter is to introduce formally the obtained tool, the second section explores all tool architecture, explaining why is it assemble in that manner and what kind of technologies this tool relies on. Finally, the third, fourth, fifth and sixth section covers all tool implementation: algorithms and structures, justifying all the roads that were taken, the implementation of each metrics and why some of them are impossible to automatically evaluate, without the help of the user.

4.1 A TOOL FOR METRIC EVALUATION

Continuing all the reasoning presented in this document so far, concerning grammars as object of study, with the knowledge and concept discussed, emerge now the **GQE - *Grammar Quality Evaluator***, which come to give support to an automatic grammar quality assessment, by performing automatic evaluation of a large set of metrics. Based on the metrics computed, any Grammar Engineering will easily be able to reason about the quality of its grammar and to improve it.

Although there exists similar tools to the one here described and already mentioned in chapter 2, the resultant GQE system is different in two crucial aspects: in one hand it is extended to deal with AGs (not only CFGs) and in other hand it will produce different results because new metrics will be considered for the assessment (notice that besides the traditional *size* metrics, the tool contribute with a new innovator set of *style* and *lexicographic* metrics).

Figure 8.: Grammar Quality Evaluator Logo.

To raise the speech once more to a formal level, the GQE is an attribute grammar compiler (processor) written in Java, and generated by ANTLR[1] from an grammar originally designed by Sam Harwell and Terrence Parr, and afterwards extended by the author with the necessary attributes and semantic rules to perform the computation of the *size*, *style* and *lexicographic* metrics that are needed. The context-free grammar used to validate the ANTLR meta-language syntax is exposed in Appendix A.

The tool reads any ANTLR Grammar (this is, any grammar written in the ANTLR meta-language) and outputs the value for each one of the metrics under consideration. The user (for sure a grammar engineering) will analyze the values provided and will be able to come up with an assessment. Easily he will be able to transform his original grammar and submit the new one for re-evaluation to understand the eventual improvement.

Before moving forward in this reasoning, it is suitable to shortly justify the main reasons why those two technologies were selected to achieve the desire solution. For ANTLR please consult Appendix B, as for Java programming language, without much explanation, was the choice because:

- it is a Object-Oriented language, free, with a very rich API and libraries, allowing the developing of application much easier, and it also helps to keep system modular, flexible and extensible;
- it is a language that author is comfortable to work with;
- it is everywhere and allow the development in tools, such as the used, IntelliJ IDEA[2];
- allow to easily link the ANTLR, the development of the application interface and the data structures, all together without any kind of translation process.

4.2 GQE ARCHITECTURE

To properly explain from scratch how GQE is architected, first this section will start by showing how exactly the GQE was assembled, as a tool, from the outside in a more abstract point of view, then will evolve to showing the inside structure as well as their components, their function and how they all interact with each others upon each user execution.

1 The ANTLR version used was 4.5.1, available for download in this link http://www.antlr.org/download.html .
2 https://www.jetbrains.com/idea/

As stated in the previous section, this application was all developed around an attribute grammar, written in ANTLR. So, before the execution of any kind of grammar, it is required to compile that grammar down into parser and lexer in Java language (for more details about how this task is produced please read section B.1 in appendix B). This way, there is no need to manually create a parser and a lexer, because it is taken advantage of the ability on ANTLR tool to automatically generate those files, from the source grammar. Even more, this work it is made only one time, when the grammar is ready and producing the desire results, before building the GQE application itself and before any execution. The ANTLR will be summon again in the execution flow for each grammar passed as argument to GQE, but more on that later.

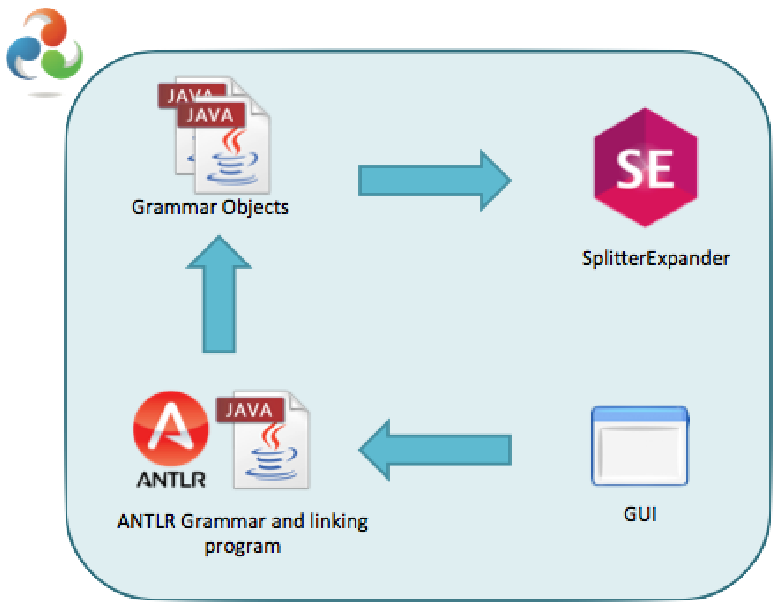


Figure 9.: Structural diagram with the four main components of GQE.

Speaking know in architectural terms, the GQE tool is composed by four main components, as shown in Figure 9: the graphical user interface or GUI, the grammar objects (Java classes), the ANTLR linking component and the SplitterExpander. It is important to notice that this architecture was assembled in a hierarchical form wherein one component will always depend on other component. Even the SplitterExpander, which is positioned on the base of this hierarchy depends on other independents programs.

The SplitterExpander component is used for the assessment of the lexicographic metrics. The ANTLR linking component is the one responsible to execute the parser and the lexer generated in the initial phase, from the Attribute Grammar designed to describe ANTLR's meta-language.

All the metrics calculation process is embedded in the attribute grammar, it is there that the Java classes, which represent the object grammar, are fed to store the value from that process as well as the call to SplitterExpander component. Finally, the GUI uses the Java classes to fetch the metrics values and display them.

4.3 GRAMMAR OBJECTS

Grammar objects are the Java classes that were created to conceptualize the notions of Context-Free Grammars and Attribute Grammars, provided by the Definition 2 and 9 respectively, from section 2.3. Basically, the idea was to create an Java abstract class to describe the object **Grammar**, then from that, create an extension of that class, with its own characteristics, to specify the object **Context-Free Grammar** and from this extension to create yet another extension to specify the object **Attribute Grammar**, just like stated in their definitions.So, in this section will be explained the purpose of this component and how is developed, as well as what is inside of each class that represents both grammar objects.

The main function of this component is from one hand to store the values outputted from an grammar recognition and from another hand to supply the user interface with the metrics already calculated. The benefits of this solution is to take all the calculation process of the grammar, thereby making the grammar more clean and readable. Another important aspect is the easiness to maintain the application as well as its own development process. In the future any update on the application can be done just by linking a value from the grammar recognition and changing a metric definition in the desire grammar object, for example.

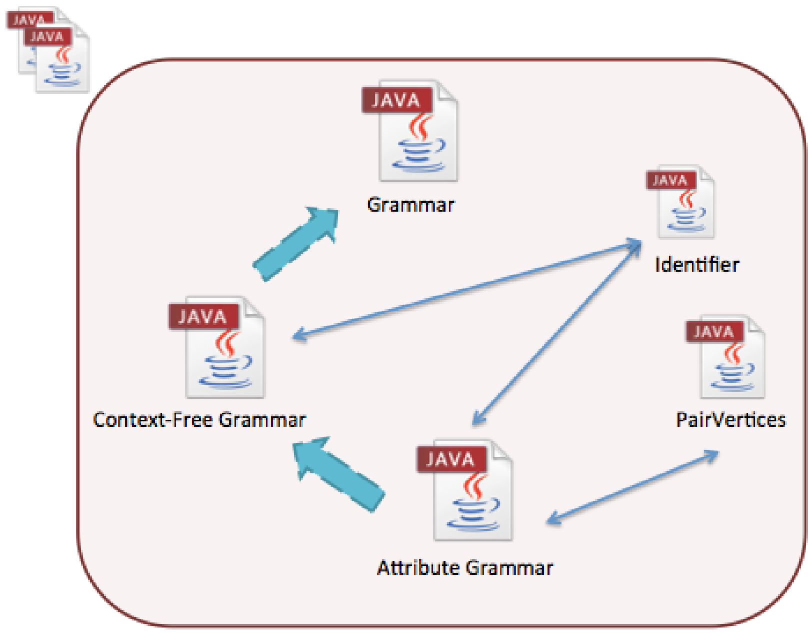


Figure 10.: Structural diagram of the Grammar Objects component.

Speaking know in structural terms, inside this component, let us take a look at Figure 10, where is shown the three main classes written in Java, each one representing a different object. Note that this objects are linked in hierarchical terms, just as was stated before: an abstract class called `Grammar`, an extension of that class called `Context-Free Grammar` and finally another extension of the last class named `Attribute Grammar`. Besides that, another two classes were implemented: `Identifier` used to instantiate the results provided by the SplitterExpander component (more on this ahead) and `PairVertices` class to facilitate the construction of the syntax/semantic complexity graphs.

Before get in more details about what is implemented in this objects, it is important to point out that the development of this objects (data structures, algorithms) was made incrementally and always regarding some factors:

- definition of each proposed metric exposed in the Subsection 2.3.3;
- easiness of data structure manipulation;
- maintenance and efficiency;

The top level object in the component is *Grammar* specified by in an abstract class with the same name. As such, this top object can not be instantiated and the purpose of its creation is to symbolize clearly the semantic between a grammar and a context-free grammar. Further more, although it is not used in this project (for know), there are other kinds of grammar, so for a future implementation this structure class can be useful.

4.3.1 *Context-Free Grammar*

The *Context-Free Grammar* class was implemented to evaluate and store the values of all proposed metrics required for assessing all input grammars of this type. In this subsection will be presented all the data structures that were used, always justifying the choice for the variables **type**, as well as, the methods that are defined in this class and how the structures were **manipulated** to achieve the desire results. Only some choices, the ones able to raise questions to the reader, will be fully dissected, such as some complex data structures and algorithms behind methods.

In the Listing 4.1 is possible to observe all the main data structure used, as well as their meaning described in the comment line of the variable declaration.

```
public class ContextFreeGrammar extends Grammar{

 private  HashSet<String> Terminals; // Terminals Symbols Table
 private  int terminals; // Number of Terminals
 private  HashSet<String> NonTerminals; // Non Terminals Symbols Table
 private  HashMap<String,String> TerminalsVariables; // TerminalsVariables Set
 private  HashSet<String> KeywordsSigns; // Keywords and Signs Set
 private  int nonterminals; // Number of nonterminals
private  int productions; // Number of Productions
private  int unit_productions;  // Number of Unit Productions
private  HashSet<String> RecSymbols; // Set of Recursives Symbols
private  double RHS; // Average number of RHS symbols
private  int RHS_max;  // Maximum RHS
private  double Alt; // Average number of alternatives
private  int Alt_max; // Maximum number of alternatives
private  int Mod;  // Number of grammatical modules imported
private  int RD;  // Number of functions from RD parser
private  int TabLL; // Dimension of Parsing Table LL(1)
private  HashMap<String,ArrayList<ArrayList<String>>> Prod; // All productions
private  int AD_LR; // Dimension of the Automata LR(0)
private  int Tabs_LR_rows;  // Dimension of the Parsing Tables LR(0)
private  int Tabs_LR_cols;  // Dimension of the Parsing Tables LR(0)
private  String rec_form; // Recursivity Form (Direct/Indirect/Mixed)
private  String rec_type; // Recursivity Type (Right/LL/Left/Mixed)
private  String notation; // Notation (BNF,e-BNF)
private  double Fan_in;  // Average number of branches In DSG
private  double Fan_out; // Average number of branches Out DSG
private  String start_symbol;  // The value of Start Symbol
private  int inline; // inline comment
private  int block; // block comment
private  int meta; // meta comment
private  HashSet<String> comments; // set of suspects for comments
private HashMap<String,Identifier> Identifiers; //Identifiers and SE results
private HashMap<String,Identifier> CommentsList; // Comments suspects
private HashMap<String,Identifier> Keywords; // List of all Keywords
private StringBuilder GrammarComments; //Comments found in grammar
```

Listing 4.1: Variables declarations of Context-Free Grammar Java class.

As it obvious, most of the variables type or why they are used raise no doubts, but it is important to justify some options:

- Terminals and Non-Terminals have a structure to store their symbols, but there is also two `int` variables to store their sizes because some of the more complex metrics use this variables in their own evaluation, so instead of always calculating the sets size it is more efficient to only do it once and store it in a value, for both sets;

- For all the Map and Set interfaces, the options to their implementation was Hash because their use is not only to storage purpose but also to help on the evaluation of other metrics. As for type of the interfaces the criteria was the definition of the metric: *Set of* ... or *List of* ..., to decide between HashSet and ArrayList, and the HashMap when was needed to link some key to his respective information;

- All results of the SplitterExpander were instantiated by the class `Identifier`;

The only variable structure that need some additional explanation is the variable `Prod`, used to store all the productions. At the first look may seem a structure to much complex, but storing the productions like this facilitated the algorithms complexity. To better understand this structure let us remember the grammar for Lists, exposed in the Example 1 and observe the data structure `Prod` of this set of productions in Figure 11. Clearly the left side symbols of the productions are the *keys* of the HashMap, which is very useful and efficient to discover the recursion form or type, for example. The *value* for each key is an $ArrayList < ArrayList < String >>$ which represent the alternatives of a rule, caring about the order. Finally, in each alternative there is an $ArrayList < String >$ storing the right hand side symbols, one more time, concerning the order in which they appear. This way the algorithms to evaluate complex metrics were, some how, not so difficult to implement.

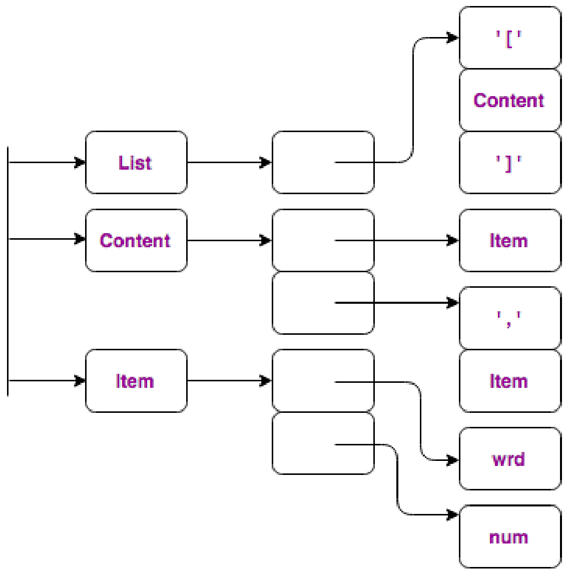


Figure 11.: Data structure for productions of the grammar from Example 1.

Concerning now the methods of this class, there are three main set of methods implemented in this class: *getters* - necessary for the intercommunication/data exchange between the ANTLR Grammar, the SplitterExpander and the User Interface component; *setters* - used to store the values in the variables, normally used by the ANTLR Grammar or some methods inside the own class[3]; and the *metrics evaluation* - methods used to evaluate some more complex metrics, to trade instantiate the data sent by the SplitterExpander and more.

Instead of exposing each method implementation to detail, a brief explanation of how each metric, used for assessing Context-Free Grammar quality, is evaluated will be presented. In other words, the algorithms behind each result achieved by GQE or as said previously, the way that the data structures are manipulated. So, for all set of proposed metrics, in the subsection 2.3.3 of the chapter 2, an

3 Also a good programming practice inside the context of Object Oriented Paradigm languages.

explanation will be given, stating either if the metric in cause is possible (how) or impossible (why) to evaluate. Some of the metrics are easy to evaluate during the grammar recognition phase, while others can only be evaluated after the input grammar is recognized.

- During the recognition phase:

  This particular subset of metrics is easy to evaluate because it can be done with an direct measurement, directly made in the grammar and in some cases in more than one place[4]. Most of the size metrics (**#T**, **#N**, **#P**, §**Alt**, §**Alt-Max**, **#Mod**) and one style metric (**notation**) are calculated based in an incremental logic related with the recognition of some element or event, like is shown in Algorithm 1[5].

Algorithm 1: Algorithm for direct measurement evaluation.

```
input  : Context-Free Grammar $CFG = (R)$ such as $R$ is a set of rules
output : Metrics evaluated

while not at end of grammar recognition do
    read parser rules specifications;
    for each production do
        read right hand side elements;
        if element recognized then
            increment element;
        end
    end
end
```

- After the input grammar is recognized:

  The metrics that fit in this situation are: the ones that needed some more information about the entire grammar to be evaluated and the lexicographic metrics.

4 This happens because some elements can appear in more than one place and form. For example, the ANTLR meta-language syntax allows that the Terminals can appear as an atom of an element in the right hand side of an alternative or as an element in the *notSet* construction, specific to ANTLR.

5 Of course some metrics need some more work, speaking in programming implementation, such as auxiliary variables or structures, flags and some more conditions, as it clearly perceptible.

Metrics such as: **#UP**, **#R**, §**RHS**, §**RHS-Max** and recursion **form/type**, are evaluated by manipulating the data structure `Prod`, as exposed by the following algorithm.

Algorithm 2: Algorithm for metrics by manipulating data structures.

```
input  : Data structure $Prod$=($K$,$V$), where $K$ is the $Prod$.KeySet() and $V$ the values set
output : Metrics evaluated

for each rule specification, $r \in K$ do
    if has only one production with one symbol then
        increment unit productions;
    end
    for each alternative, $alt \in V(r)$ do
        if alternative is empty then
            if exists right recursion in r then
                $rightRecursion \longleftarrow true$;
            end
            if exists left recursion in r then
                $leftRecursion \longleftarrow true$;
            end
            if exists right recursion LL in r then
                $rightRecursionLL \longleftarrow true$;
            end
        end
        for each element, $elem \in alt$ do
            increment $RHS$;
            if element is equals to r then
                increment recursive symbols;
                $directRecursion \longleftarrow true$;
                if elem is the first symbol from the left then
                    if exists empty alternative in rule r then
                        $leftRecursion \longleftarrow true$;
                    end
                else
                    if exists empty alternative in rule r then
                        checkLL($elem$,$alt$);
                    end
                end
            else
                if element is a Non-Terminal then
                    checkIndirectRecursion($elem$,$r$);
                end
            end
        end
    end
end
```

With the intention of not extend the previous algorithm and thereby making it, more difficult to read and less perceptive, some functions has been added. The algorithms used in these functions will are shown in the Algorithms 3 and 4.

Algorithm 3: Algorithm for finding indirect recursion pattern.

```
input  : $(elem, r)$ - element in the RHS of rule $r$.
output : check Indirect Recursion.

for each alternative in elem rule specification do
    for each symbol in alternative do
        if symbol is equal to $r$ then
            increment recursive symbols;
            $indirectRecursion \longleftarrow true$;
        end
    end
end
```

Algorithm 4: Algorithm for finding LL(1) recursion pattern.

```
input  : $(elem, alt)$ - element and the set of productions of $r$.
output : check LL(1) Recursion.

for each rule specification do
    if rule contains one alternative equal to $alt$ and the LHS is different from $elem$ then
        $rightRecursionLL \longleftarrow true$;
    else
        $rightRecursion \longleftarrow true$;
    end
end
```

After processed the Algorithm 2 some other metrics are know evaluated, such as: **#RD**, §**TabLL**, §**TabsLR**, **FanIn**, **FanOut**, because they are calculated at the expense of others, already valued.

The only metric left, aside the lexicographic metrics, is §$\mathcal{DA}$**-LR**, known as the dimension or number of states of the Deterministic Automaton LR(0). This metric is not so easy to evaluate without actually building the deterministic automaton, but it is deductible that the transition function between states is related with the recognition of a certain symbol, with some exceptions. So, the method used to number of states count was implemented regarding the following premises:

- the number of states corresponds to the number of right hand side symbols of all productions. So, every time that a symbol in an alternative is recognized, the count is incremented, except when:
  1. the alternative is empty, $X_0 \rightarrow \varepsilon$;

2. two or more alternatives in the same rule specification(X_0) start with the same sequence of symbols,
$$X_0 \rightarrow X_1; X_0 \rightarrow X_1 X_2;$$
3. exists one rule(X_1) with an alternative where left recursion is present and another rule(X_0) with an alternative that start with X_1
$$X_0 \rightarrow X_1; X_1 \rightarrow X_1 \ldots;$$

– the final state must be counted, that is why the count starts in 1, because we have to considerate the state when all grammar is recognized. The symbol that is between the initial state and the final state is the initial symbol of the grammar;

The lexicographic metrics are evaluated in the end of the recognition phase because they require a call to the SplitterExapander component. This call is made by sending data to an external perl program which takes always some time. So, instead of making that call every time that an identifier is recognized, all the variables that need the SplitterExpander are joined in the end, and the process communication is done.

The **clear identifiers** metric is achieved by sending two different sets(terminals and non-terminals) to the SplitterExpander, which will verify if those identifiers derives from the concept name or not. For more information about how all this process is done, please check Section 4.5.

The first step back found in the implementation of these metrics was evaluating **reserved-words and clear signs** metric. Evaluating the reserved-words is easy because follows the same method used for the identifiers. The difficulty arises when we talk about clear signs, not because they are hard to recognize but because it is very complex or even impossible to say if they are clear or not, automatically. Imagine that a list is specified in a grammar and the separator used between the elements is the comma(',') sign. For us, humans it is obvious that the comma sign can be considered clear and enlightening, but for a machine to do it automatically is not so obvious.

Another metric that was not implemented was **terminal-variables flexibility**. This metrics is related with the lexer rules for the terminals variables and depend to much on what the grammar format allows to specify the tokens. ANTLR format is very extensible in this case and allows constructions such as regular expressions, fragments, lexer modes, recursion and others to specify the right hand side of a lexer rule, which makes the automatic evaluation very difficult.

Finally, to evaluate the **kind of comments** that the language specified by some grammar allows, it was taken advantage of the SplitterExpander. A list of possible or provable identifiers for inline, block and meta-information comments was created, and embedded in the SE source code. All lexer rules that can contain the specification of those three kind of comments are crossed with that list.

4.3.2 *Attribute Grammar*

Following the train of concepts presented in the start of this section, emerge know the Attribute Grammars. As defined before, in this document, attribute grammar are an extension of context-free grammar combined with a set of attributes and their semantic meaning. To represent this concept, in this application, a Java class was created to evaluate and store the values of all proposed metrics required for assessing know, attribute grammars.

Once more, the intention is to give an idea of the structures and algorithms used to secure such assessment, always justifying the paths that were taken, how each metric is evaluated and if is not why.

```
public class AttributeGrammar extends ContextFreeGrammar{

 private int attributes; // Number of Attributes
 private int syn_attr; // Number of Synthetized Attributes
 private int inh_attr; // Number of Inherited Attributes
 private int CR; // Number of Calculation Rules
 private int complex_attr; // Number of attributes with structured type
 private String values_aggreg; // Form of values aggregation
 private String values_accum; // Form of values accumulation
 private HashMap<String,Integer> Attr; // Each attribute and his complexity
 private HashMap<String,ArrayList<PairVertices>> LDG; // Local Dependencies Graph
 private HashMap<String,ArrayList<String>> Syn;  //synthetized att per NT
 private HashMap<String,ArrayList<String>> Inh; //inerited att per NT
 private double ag_Fan_in;  // Average number of branches In LDG
 private double ag_Fan_out;  // Average number of branches Out LDG
 public HashMap<String,Identifier> attributesIds; //Att. and their SE results
```

Listing 4.2: Variables declarations of Attribute Grammar Java class.

Relatively to this Attribute Grammar class, the variables and their respective data types are shown in Listing 4.2. Here, there is not much to be told, because all the data types are easily understandable towards its purpose and what it have to store. Just like before, the idea is to make a clear notion of each metric and use this declarations to facilitate the calculation process and its efficiency.

Taking into account, once more, the set of proposed metrics to assess, this time attribute grammars, defined in subsection 2.3.3, a short explanation of each metric implementation will be provided now. The same division can be made about the metrics calculation process:

- During the recognition phase:

  All metrics proposed to assess attribute grammars size(**#A**,**#IA**,**#SA** and **#CR**) are inserted in this phase and all share the same logic towards its evaluation, shown already in Algorithm 1. Note that the number of context conditions(**#CC**) and the number of translation rules(**#TR**) were not implemented because they are not calculable in the ANTLR format. Consequently, form metrics **semantic restriction scheme** and **translation scheme** are also impossible to evaluate, for now and for this format. Although, in the future this metrics can be implemented for a different grammar specification format and demonstrate their purpose.

Thanks to the addition of the Java code specification in the ANTLR meta-language grammar, exposed in Appendix A, was possible to evaluate some more metrics. For **attributes complexity** the followed the logic is presented Algorithm 5.

Algorithm 5: Algorithm for attributes evaluation.

```
input  : Attribute Grammar
output : Metrics evaluated

while not at end of grammar recognition do
    read parser rules specifications;
    for each rule do
        read attributes and semantic information;
        for each attribute recognized do
            read attribute type;
            if attribute has complex type then
                increment attributes complexity;
            end
        end
    end
end
```

Now, to assess the **calculation scheme** for writing calculation rules it was needed to evaluate the form of values aggregation and the form of values accumulation. This idea of presenting schemes to aggregate or accumulate values to build the semantic value of a grammar may seem ingrate because there is a lot of valid ways to do it. The method used to identify **aggregation values patterns** was searching for a generic aggregation function, with the following form:

$$X_0 \rightarrow X_1 \dots X_n; \{X_0.s = genericFunction(X_1.a, \dots, X_n.a)\}$$

After this *genericFunction* being found, some invariants must be hold, to be recognized as a aggregation pattern:

- the arguments list of the *genericFunction* can not be empty;
- the attribute s from X_0 must have complex type;
- each *genericFunction* argument must return and attribute from the Terminals or Non-Terminals presented in the productions of X_0 specification.

As for identifying **accumulation patterns** no effort was needed. if the grammar only use synthesized attributes to pass the information around the derivation tree, then we are in the present of a purely synthesized pattern. In the other hand if only inherited attributes are used then we have a purely inherited pattern.

- After the input grammar is recognized:

  Once again, the lexicographic metrics to assess, this time, attributes are found here. Both metrics, **clear identifiers for attributes** and **attributive operations**, follow the same logic explained before for lexicographic metrics in Context-Free Grammars.

  Finally, the semantic complexity metrics(**FanIn** and **FanOut**) were calculated with some difficulty level. Besides the difficulty to build the Local Dependencies Graph between attributes, arise the task of identifying the attributes used for another attribute construction. Algorithm 6 shows how this graph were built and from that the number of branches that go in and out on attributes are easy to count.

Algorithm 6: Algorithm for semantic complexity evaluation.

```
input  : Attribute Grammar
output : Metrics evaluated

for each calculation rule do
    for each statement expression do
        if is an attributive expression, with the purpose $A = B$ then
            if $A$ is an rule attribute and expression $B$ contains attributes of the RHS symbols then
                increment Local Dependencies Graph;
            end
        end
    end
end
```

  All the work lies in this *attributive expressions* presented in statements, because these expressions can take various forms, for example: $a = b$, $a = f(b)$ or $a++$, but all with the same purpose: to give a value at some element. All possible expressions are specified in grammar rule `expression`(line 518 in Appendix A) but only a part of them allow attributions. The logic is to identify these attributions, and by consequence if the attributive element is an attribute from the grammar and if other attributes, from the grammar, are used in these attributions.

Outside of this division are two form metrics **style of the language** and **language specificity** because they depend exclusively on the grammar format. In the ANTLR format case, attributive operations are written in standard programming language Java, oriented to the objects paradigm.

4.4 ANTLR LINKING COMPONENT

This component is an important part of all GQE structure, not because of the complexity presented in it, but because of its function and what it represents. Therefore, this section is dedicated to this component exposition, saying what is in it and explaining some of the process.

The main function of this component, as detectable from this section name, is linking all the ANTLR files with the other components, more precisely the Grammar Objects and the User Interface. At the beginning of the input grammar recognition process, a Java object is created(Context-Free Grammar or Attribute Grammar) and initialized. In the end the User Interface retrieves this object from this component and display the metrics.

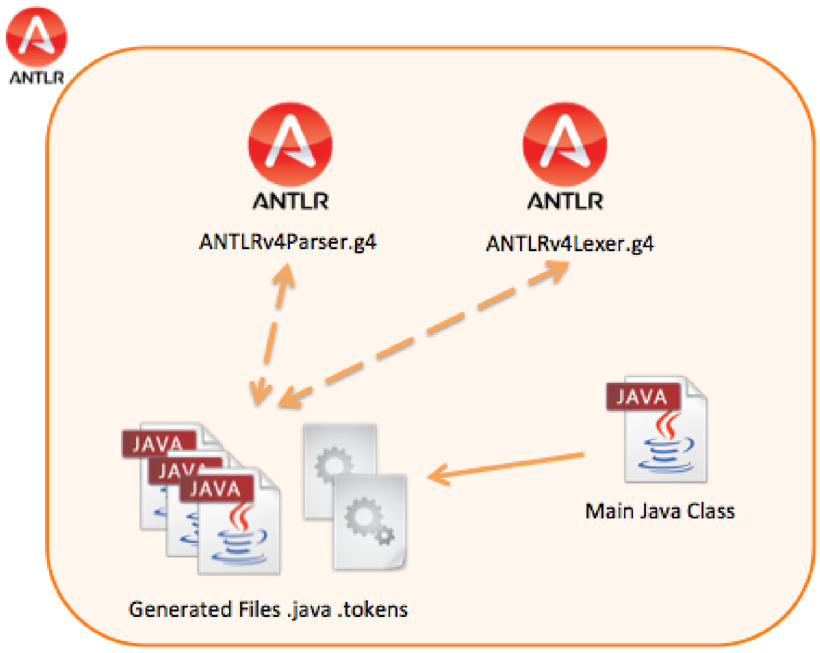


Figure 12.: Structural diagram of the ANTLR linking component.

Figure 12 shows the structure of this component, with the separated grammar files: parser and lexer, the generated files from running ANTLR on the parser and on the lexer, and finally a main Java class to tie them all together.

As stated already in 4.2 from the `ANTLRv4Parser.g4` and `ANTLRv4Lexer.g4` files, the ANTLR tool will generate a *.tokens* and a *.java* file for each one. The purpose of the main class is to create a lexer and parser object(with the help of the generated files) and executing the parsing process. This main class is similar to the Test class defined in Figure 34, the only differences are that the start symbol of the grammar is **grammarSpec** and the input stream come from an input file, selected by the user.

4.5 SPLITTER EXPANDER

The *SplitterExpander* is a program developed in Perl programming language responsible to help on the assessment of the lexicographic metrics, for both Context-Free Grammars and Attribute Grammars, defined in the subsection 2.3.3, chapter 2.

Recalling the definition 12, in which it is established the validity of an Identifier by verifying if it *derives* from a concept name or not, this program was created with the desire to perform three main tasks:

1. to **split** the Identifier into logical pieces, if some splitting pattern were found or both, such as the *CamelCase* technique[6] or the *'_' (underscore)* separator technique[7];
2. to **expand** the Identifier to the correct concept name, if the Identifier is a prefix or if it was divided in pieces to expand each one to the correct concept name;
3. to validate the syntax of the Identifier;

At first sight, this goal to evaluate the correct structure and to expand an Identifier into the precise concept name for which was created, may seem bold, because this task appear to depend on the logic deduction of a human being, but the aim is to approximate the program results to the evaluation of the human reasoning.

Just to prove that all this wanted behavior it is not easy to accomplish let us take a look to some examples, in the range of possibilities that could appear, demonstrating the source Identifier and the expected result from processing it.

Example 5. *The most simple case is from the Identifiers that has no need to be splitted neither expanded. The only restrictions is that they have to be composed by more than one letter and have to belong to English dictionary.*

Example 6. *Other possible case come from the Identifiers that need to be splitted but do not need to be expanded.*

Example 7. *Some Identifiers or pieces of Identifiers that were splitted need to be expanded. In some cases the expansion of such sequences of characters are easy to predict.*

But different from all the previous examples, are the Identifiers in which the expansion is not that easy to predict.

6 Classic technique in programming languages used to separate different words in a Identifier. Very clean and legible for those who are reading the code and trying to understand the purpose of some methods or variable. Some examples are: `DestAddrLst` which can be splitted and expanded to *Destination Address List* or `incrementTerminals` that can be splitted in *Increment Terminals*, with no need for expansion.

7 Another classic Identifiers separation technique, but this time instead of using the Camel Case to highlight the separation, the meta character underscore '_' is used. Few examples are: `number_of_productions`, `print_Tab` for *print table*.

Example 8. *Identifiers that the expansion is not so predictable, for example prefixes or just letter in the middle of an Identifier.*

All the tasks listed before are achieved by the SplitterExpander with the help of two other programs: a perl module called *Lingua::IdSplitter*[8] and other natural processing language tool named *Wordnet*[9], just as shown by Figure 13. For more details about this natural languages tool, please see Appendix C.

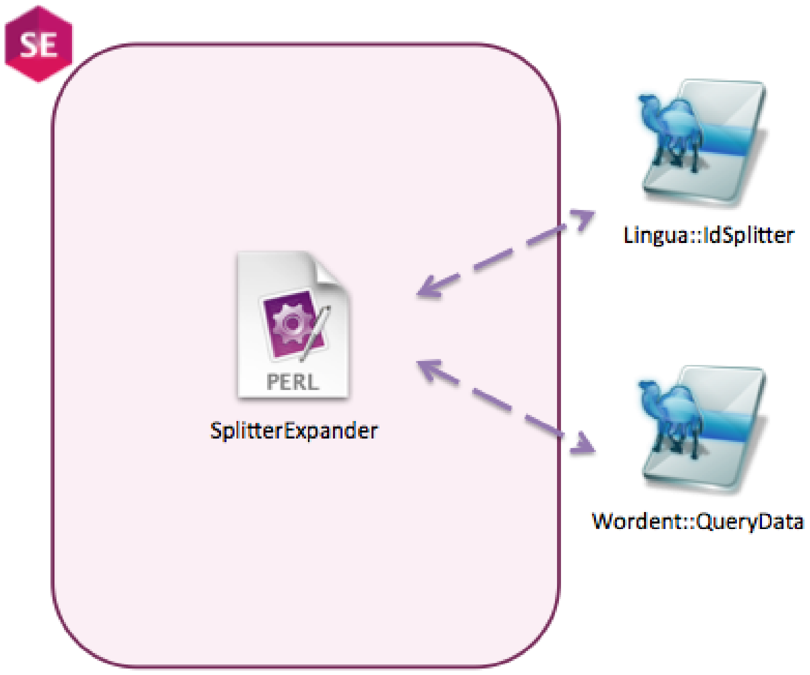


Figure 13.: Structural diagram of the SplitterExpander component.

The perl module Lingua::IdSplitter was edited to better suit the integration with the rest of the application and to produce other kind of results. He is responsible to identify the splitting patterns (CamelCase and '_' separator), then split the Identifier into smaller identifiers and then check the semantic validity of that identifier. Another feature of this module is that have implemented some custom dictionaries for some easy expansions.

The other perl module presented in Figure 13 is Wornet::QueryData. This module is only used as an interface for the already introduced tool *Wordnet*, because it allows a direct query context with all the data relations that *Wordnet* provides. It is more clear and efficient to work with a perl module in perl programming language then to build some mechanism to contact with the stand alone tool it self.

8 https://github.com/nunorc/Lingua-IdSplitter
9 https://wordnet.princeton.edu/wordnet/

With the intention of going far ahead in the expansion of some complex Identifiers, such the ones provided by the example 8, the idea was to isolate those identifiers and apply on them three heuristics that could help to make a rational decision for theirs expansion.

- Heuristic One **[Documentation]** - Clues about some identifiers most probable expansion are looked in the documentation files of the grammar, if they exists;
- Heuristic Two **[Comments]** - During the recognition phase of the input grammar, all the recognized comments are joint together so that they can be sent to the SplitterExpander. This comments may be useful, for some complex expansions;
- Heuristic Three **[Grammar Domain]** - Grammar Domain can be specified in the Interface by the user, if not, the application will assume that the top level domain of the grammar is the grammar name. With this domain, and with the help of Wordnet, a list of related words (various relations) is presented. Once again, the idea is to expand some ambiguous identifiers with this domain related words.

Of course all these heuristics starts from the notion that all the grammars designers are good, this is, they comment correctly the grammars, the rules and the productions, they have documentation files in the same directory of the source grammar and they give a clear identifier to the name of the grammar. Even so, if it is possible to apply at least one of the heuristics to the grammar, the result could be very helpful and determinant to the correct expansion of some of the grammar Identifiers. More, if appear two or more different possibilities for some identifier expansion, this heuristics may give us some decision about which is the most probable possibility, for that identifier, for that grammar, in that context.

4.6 GQE INTERFACE

This section present the visual component of the GQE application. It contains all the information about the User Interface and answer to some questions, such as: what is the purpose of this component, in which technologies was built, what resources consumes and the output it produce.

The objective of this component, as is deductible by its name, is to perform the interface between the GQE application and the user, for sure a grammatical engineer. This interface can be seen as the result of all the work made by all the others components. It was only one simple task: to show the results of the computed metrics, responsible to assess the input grammar.

Although, considering this as a simple component, it was built upon some desire aspect and behavior, characteristics that should be share by all interfaces. The intention was to design an efficient interface, easy to manipulate by any user, without to much windows or panels, providing results in very few steps and aesthetically pleasant.

For this simple component, to achieve this objectives, it was structured like shown in Figure 14. In the total, the User Interface has three main interaction elements: a Presentation Panel, a Input options panel and a Results display panel. The Presentation Panel is used as an introduction for the application, presenting GQE and informing the user that the program was been initialized. The Input options panel requires more interaction with the user, for choosing the input grammar file and to assess the grammar with some options, such as the type(CFG or AG), the format(for the future because know it only accept ANTLR format) and more. Finally, the Result Panel is used to display the metrics evaluation and the quality conclusions inferred towards the input grammar.

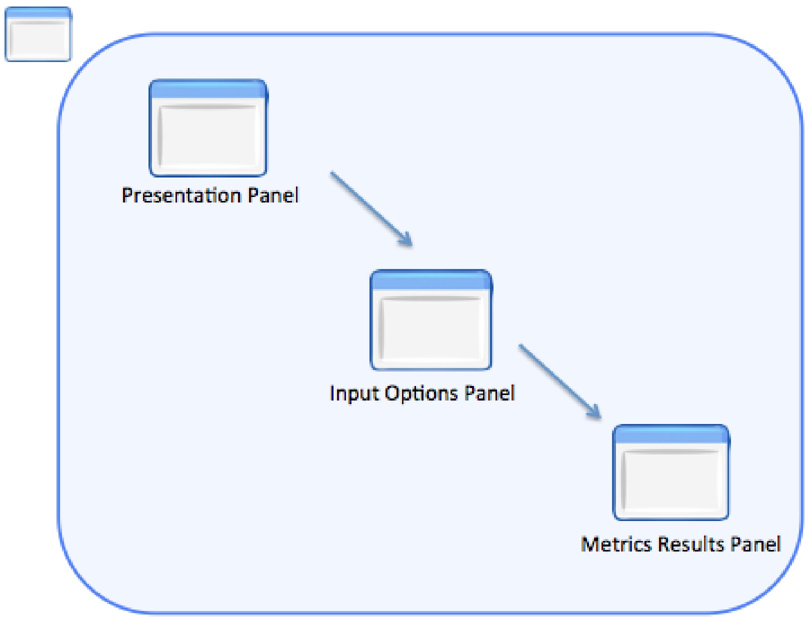


Figure 14.: Structural diagram of the User Interface component.

This interface was built with the help of the Swing Java technology, a GUI widget toolkit for Java. In other words, Swing is an API suit that allows the design of interfaces by creating and manipulating, already defined, graphical components for Java programs[10]. The benefits of this option are obvious, because it provides total compatibility with the others GQE components built in Java programming language and allows an easy way to build and maintain, nice interfaces.

4.7 EXECUTION FLOW

Now, that all the four components have been introduced and the purpose of each one exposed, comes the section responsible to explain all the activity performed by GQE. Since the moment that the user indicates the input grammar to be consumed by the tool, until the moment that the quality report is displayed on the interface, a lot of processes are executed in the background.

10 For more information about this technology, please consult http://docs.oracle.com/javase/tutorial/uiswing/.

All this process starts when the user execute GQE and select the input grammar. Right after this selection, the tool will validate syntactically the grammar in question, to see if respects the form of the ANTLR format. As is obvious, all this activity happens in the Interface component, responsible to listen the type of grammar that the user wish to assess and therefore sending it along with the *file_path* to the ANTLR linking component.

With the type of grammar and the already generated files (from compiling the used grammar to describe ANTLR meta-language) a new instance of parser and lexer object are created using as input the received *file_path*. The recognition phase of the input grammar starts from the moment that this component execute the *parse()* procedure.

Once triggered this recognition process a new Grammar Object(Context-Free Grammar or Attribute Grammar) is created according the type previous selected by the user. Trough the methods implemented in those objects, the data structures there declared are fed every time a new symbol or constructor of the grammar is recognized. Some of the metrics evaluation happens directly in the recognition moment and others are only evaluated once all the file is accepted, as explained before. Specifically, the set of lexicographic metrics is very important because it is the one that communicates with the SplitterExpander component.

All the data exchange between the Grammar Object and the SplitterExpander is done by sending JSON files. This option is taken because JSON files are simple to create/read, efficient for the task proposed and because they can be used, in the future, for sending information for others external systems. After verifying the clarity of the identifiers a new JSON file is sent in return, for that all the results can be stored.

After all the metrics calculation, the Grammar Object is retrieved by the Interface component, updating the display panel with all the information stored in the data structures, including the quality report, which present the desire final output. Figure 15 shows this flow between components.

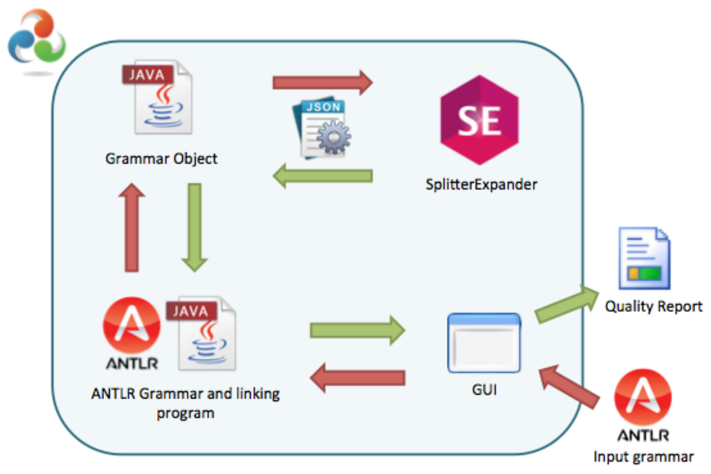


Figure 15.: Execution flow of GQE.

5

GQE MAIN RESULTS

Following the trend of this dissertation, comes now the chapter responsible to show the real application of the main result, in this case, the real application of the GQE tool introduced in the previous chapter. As stated before in this document, this tool is inserted in the Grammar Engineering field of study therefore it is there that the product of this investigation have its real application and provides an increment such as this knowledge system to grammars.

So, the purpose of this Main Results chapter is to present the benefits of using this tool in two types of grammars, more precisely Context-Free Grammar and Attribute Grammars. In order to achieve that, different grammars, in quality terms, will be used as Case Studies, showing the different results displayed by GQE for each one of them.

The approach chosen was to use one case study of Context-Free Grammar and expose all the results displayed by GQE, in an complete way, and repeat the process with another case study but this time of Attribute Grammar. Then, promptly, some pieces of different elements used in grammars, that have a direct impact in the quality of the grammar, will be shown, confronting them with the respective outputted result. Some verification will be made to check if the tool is behaving desireously according different inputs.

5.1 CFG ANALYSIS: LISP LANGUAGE

```
grammar Lisp;

lisp : sExp ;

sExpList : sExp sExpList
         |
         ;

sExp : NUM
     | WRD
     | '(' sExpList ')' ;
```

Listing 5.1: A Context-Free Grammar example for specifying Lisp language.

The grammar listed in Listing 5.1 will be used as case study for Context-Free Grammars in this section. It says that a Lisp sentence is a symbolic expression(*sExp*) and that a symbolic expression is an atomic value - number(*NUM*) or a words(*WRD*) - or a list of symbolic expression(*sExpList*) between parentheses.

The result of analyzing this grammar in GQE tool will be presented first by the metrics evaluation, and then by the assessment report, both performed automatically by GQE. This automated process is successfully done by this tool thanks to the implementation of the algorithms, exposed in Subsection 4.3.1.

- **Metrics Evaluation**

  Figure 16 shows the panel in GQE interface responsible for displaying the calculated value for each grammar size metric, defined in Table 4. As is possible to verify the results obtained from GQE are correct. The grammar have in fact:

  - 4 terminal symbols - *NUM*, *WRD*, *'('* and *')'*;
  - 3 non-terminal symbols - *lisp*, *sExp* and *sExpList*;
  - 6 productions - $p0$ in Line 3, $p1$ in Line 5, $p2$ in Line 6, $p3$ in Line 9, $p4$ in Line 10 and $p5$ in Line 11;
  - no unit productions;
  - 2 recursive symbols - *sExp* and *sExpList*;
  - an average number of right hand side symbols of 1,3 $= (1+2+0+1+1+3)/6$;
  - a maximum right hand side of 3 symbols in production $p5$ Line 11;
  - an average number of alternatives of 2 $= (1+2+3)/3$;
  - a maximum of 3 alternatives in *sExp* specification;
  - and no imported modules.

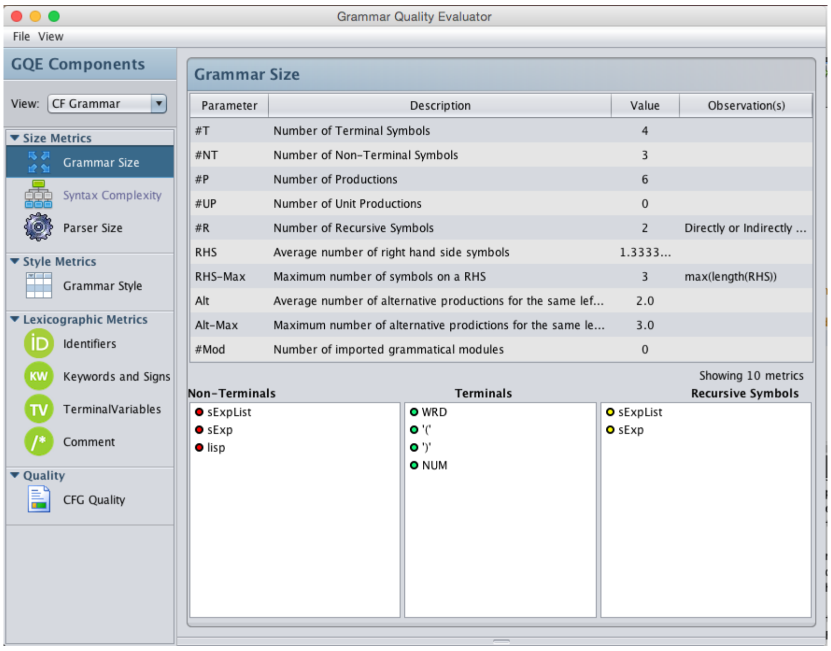
  For the subset of syntax complexity metrics defined in Table 5, the outputted results by GQE, are presented in Figure 17, such as the resultant Dependencies Graph between Symbols[1]. Here the metrics evaluation are also accordingly to the desire:

  - a FanIn of 2,6 $= (1+2+5)/3$;
  - and a FanOut of 1,1 $= (0+2+1+1+1+2+1)/7$;

  Relatively now, to the last subset of size metrics - Parser size - listed in Table 6, it is possible to see the results in Figure 19. By analyzing them, once more the tool proven to be accurate:

  - 7 functions in the Recursive Descendant Parser $= (4+3)$;
  - the dimension of Parsing Table LL(1) $= (3*(4+3))$;
  - 10 states for the Deterministic Automaton LR(0), just as shown in Figure 18;

1 GQE interface shows this graph to the user because it gives a different perspective of how complex the grammar in question, is.



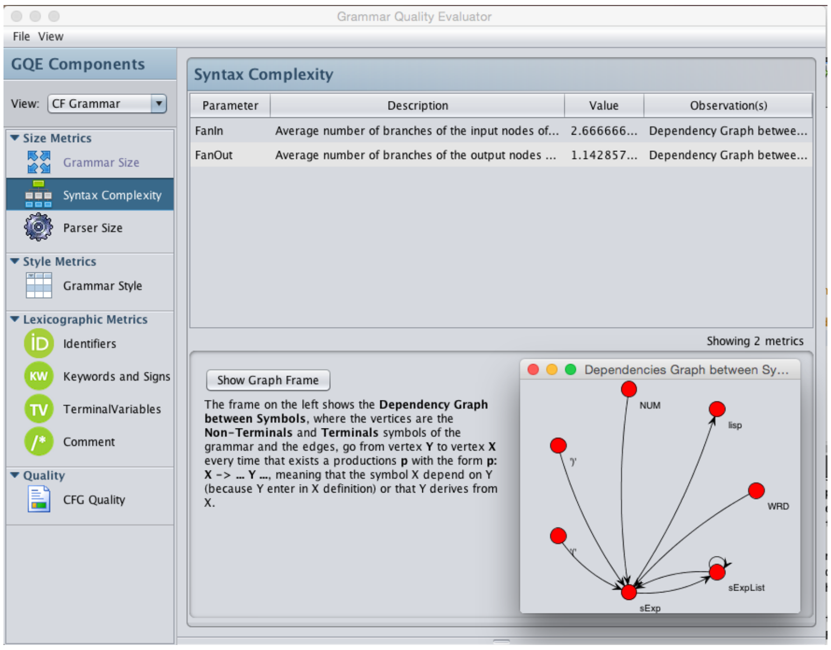
Grammar Size

Parameter	Description	Value	Observation(s)
#T	Number of Terminal Symbols	4	
#NT	Number of Non-Terminal Symbols	3	
#P	Number of Productions	6	
#UP	Number of Unit Productions	0	
#R	Number of Recursive Symbols	2	Directly or Indirectly ...
RHS	Average number of right hand side symbols	1.3333...	
RHS-Max	Maximum number of symbols on a RHS	3	max(length(RHS))
Alt	Average number of alternative productions for the same lef...	2.0	
Alt-Max	Maximum number of alternative prodictions for the same le...	3.0	
#Mod	Number of imported grammatical modules	0	

Showing 10 metrics

Non-Terminals	Terminals	Recursive Symbols
● sExpList	● WRD	● sExpList
● sExp	● '('	● sExp
● lisp	● ')'	
	● NUM	

Figure 16.: Computed grammar size metrics for CFG specifying Lisp language.



Syntax Complexity

Parameter	Description	Value	Observation(s)
FanIn	Average number of branches of the input nodes of...	2.666666...	Dependency Graph betwee...
FanOut	Average number of branches of the output nodes ...	1.142857...	Dependency Graph betwee...

Showing 2 metrics

The frame on the left shows the **Dependency Graph between Symbols**, where the vertices are the **Non-Terminals** and **Terminals** symbols of the grammar and the edges, go from vertex **Y** to vertex **X** every time that exists a productions **p** with the form **p: X -> ... Y ...**, meaning that the symbol X depend on Y (because Y enter in X definition) or that Y derives from X.

Figure 17.: Computed syntactic complexity metrics for CFG specifying Lisp language.

– and $(50;30)$ for the dimension of Parsing Tables LR(0) $= (10*(4+1);10*3)$.

The set of style metrics is divided in three metrics: form of recursion, type of recursion and notation, defined in Table 7, Table 8 and Table 9 respectively, can be seen in Figure 20. Clearly:

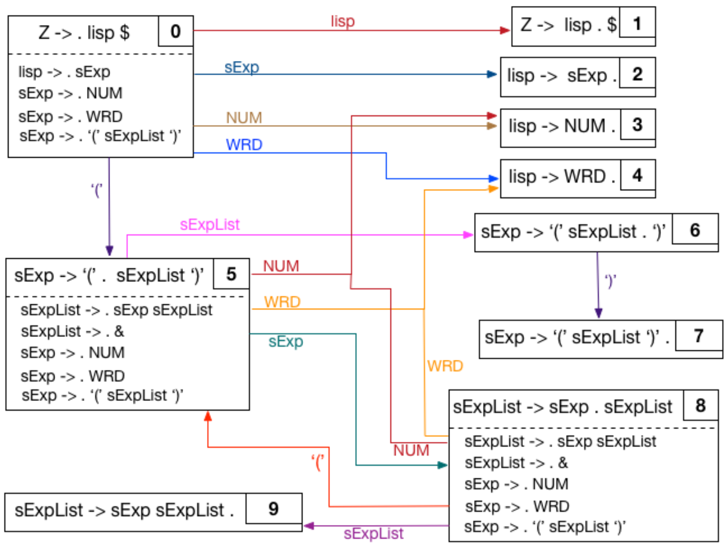
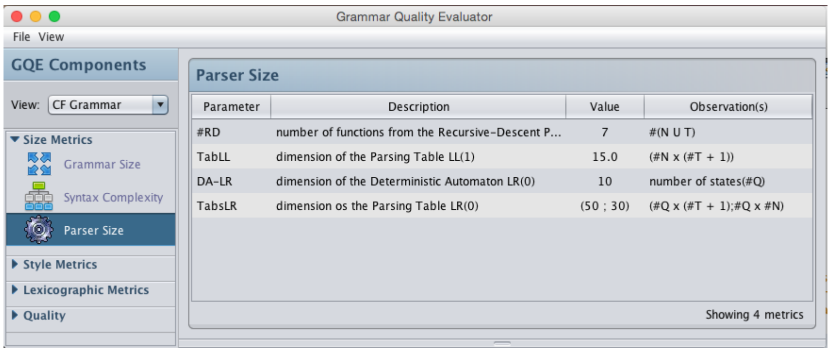

Z -> . lisp $ 0
lisp -> . sExp
sExp -> . NUM
sExp -> . WRD
sExp -> . '(' sExpList ')'
Z -> lisp . $ 1
lisp -> sExp . 2
lisp -> NUM . 3
lisp -> WRD . 4
sExp -> '(' . sExpList ')' 5
sExpList -> . sExp sExpList
sExpList -> . &
sExp -> . NUM
sExp -> . WRD
sExp -> . '(' sExpList ')'
sExp -> '(' sExpList . ')' 6
sExp -> '(' sExpList ')' . 7
sExpList -> sExp . sExpList 8
sExpList -> . sExp sExpList
sExpList -> . &
sExp -> . NUM
sExp -> . WRD
sExp -> . '(' sExpList ')'
sExpList -> sExp sExpList . 9
lisp
sExp
NUM
WRD
'('
sExpList
')'

Figure 18.: Deterministic Automaton LR(0) for CFG specifying Lisp language.



Parser Size

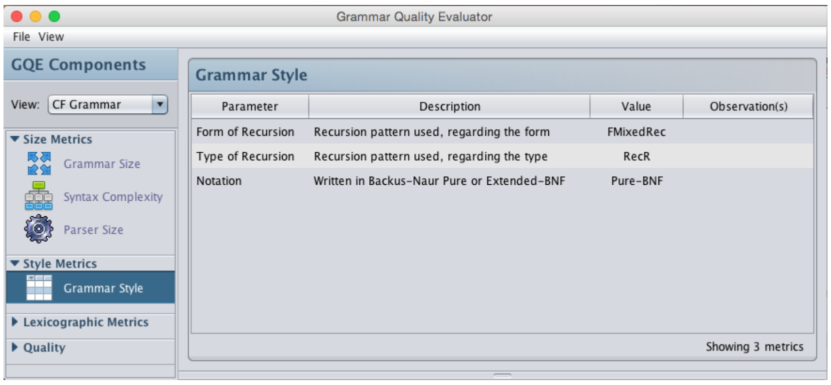
Parameter	Description	Value	Observation(s)
#RD	number of functions from the Recursive-Descent P...	7	#(N U T)
TabLL	dimension of the Parsing Table LL(1)	15.0	(#N x (#T + 1))
DA-LR	dimension of the Deterministic Automaton LR(0)	10	number of states(#Q)
TabsLR	dimension os the Parsing Table LR(0)	(50 ; 30)	(#Q x (#T + 1);#Q x #N)

Showing 4 metrics

Figure 19.: Computed parser size metrics for CFG specifying Lisp language.

- direct recursion is present in production $p1$ Line 5 and indirect recursion in production $p5$ Line 11;
- the type of direct recursion is right recursion;
- and all production are written in pure-BNF.

Finally, for the lexicographic metrics set only the results of the Clear Identifiers metric, revealed in Table 10, are shown in Figure 21. The option of showing only this metric of the referred set, was taken because it is the only element that have a direct effect on Context-Free Grammars quality, just like state already in Table 2. In the light of what was imposed by Definition 12, the results provided by GQE are correct once more:

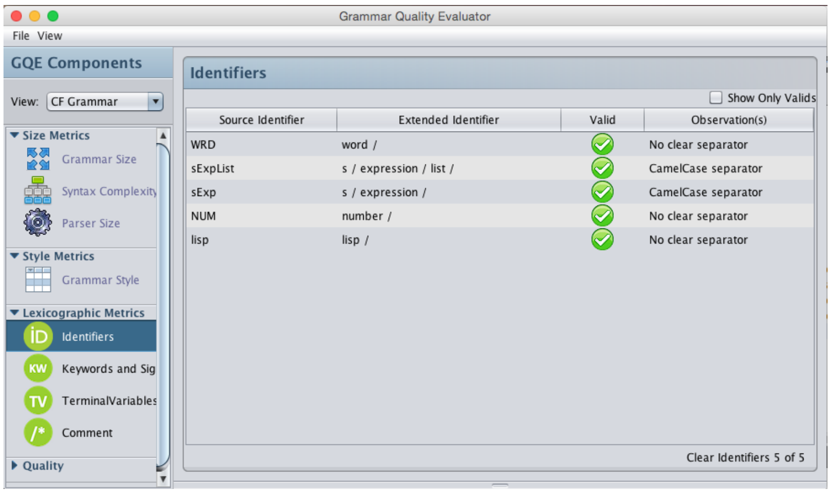


Parameter	Description	Value	Observation(s)
Form of Recursion	Recursion pattern used, regarding the form	FMixedRec	
Type of Recursion	Recursion pattern used, regarding the type	RecR	
Notation	Written in Backus-Naur Pure or Extended-BNF	Pure-BNF	

Showing 3 metrics

Figure 20.: Computed style metrics for CFG specifying Lisp language.

- the identifiers for the Non-Terminals (*lisp*, *sExp* and *sExpList*) are considered clear, because all of them *derive* from the concept names - *Lisp*, *Symbolic Expression* and *Symbolic Expression List*;
- the 2 terminals-variables (*NUM* and *WRD*) are also considered clear, for the same reason - *Number* and *Word*.



Source Identifier	Extended Identifier	Valid	Observation(s)
WRD	word /	✓	No clear separator
sExpList	s / expression / list /	✓	CamelCase separator
sExp	s / expression /	✓	CamelCase separator
NUM	number /	✓	No clear separator
lisp	lisp /	✓	No clear separator

Clear Identifiers 5 of 5

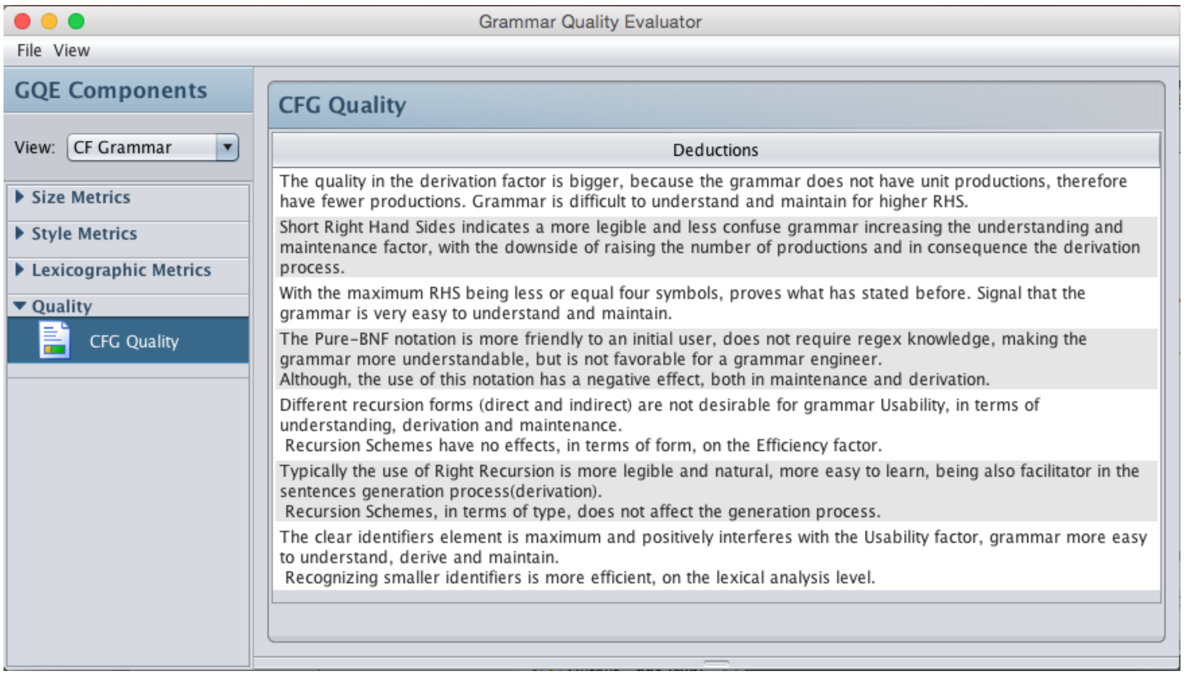
Figure 21.: Computed lexicographic metrics for CFG specifying Lisp language.

Note that the GQE interface also shows the expansions for each identifier and, in the case of a multiple terms identifier, the separator used(*Observation(s)* field).

- **Assessment Report**

  This assessment report, performed automatically by GQE, on Context-Free Grammars is the great benefit for the users of this application. It provides quality assumptions allowing the final user to reasoning about his grammar and possibly to also compare it to another.

Relatively to the grammar in study, the quality report displayed by this tool is exposed in Figure 22. Looking more carefully to this report and taking into account what has been discussed over subsection 2.3.2, it is perceptible that all generated quality assumptions, towards the grammar, are in conformity to the truth.



Deductions
The quality in the derivation factor is bigger, because the grammar does not have unit productions, therefore have fewer productions. Grammar is difficult to understand and maintain for higher RHS.
Short Right Hand Sides indicates a more legible and less confuse grammar increasing the understanding and maintenance factor, with the downside of raising the number of productions and in consequence the derivation process.
With the maximum RHS being less or equal four symbols, proves what has stated before. Signal that the grammar is very easy to understand and maintain.
The Pure-BNF notation is more friendly to an initial user, does not require regex knowledge, making the grammar more understandable, but is not favorable for a grammar engineer. Although, the use of this notation has a negative effect, both in maintenance and derivation.
Different recursion forms (direct and indirect) are not desirable for grammar Usability, in terms of understanding, derivation and maintenance. Recursion Schemes have no effects, in terms of form, on the Efficiency factor.
Typically the use of Right Recursion is more legible and natural, more easy to learn, being also facilitator in the sentences generation process(derivation). Recursion Schemes, in terms of type, does not affect the generation process.
The clear identifiers element is maximum and positively interferes with the Usability factor, grammar more easy to understand, derive and maintain. Recognizing smaller identifiers is more efficient, on the lexical analysis level.

Figure 22.: Assessment report for CFG specifying Lisp language.

Concerning now the quality of the language generated by this Context-Free Grammar, can only be stated that:

- it will not be so easy to understand the language;
- if the respective parser is well chosen and implemented, its recognition can be efficient;
- to support what has listed before is the fact that the grammar does not allow any kind of comments in the generated language, complicating the understanding and not degrading the efficiency;
- the language expressiveness will not be facilitated for the absent of keywords, but otherwise, the scalability will be ensured in terms of writing and processing large programs.

This permanent relation between the quality of the grammar and the quality of the language generated by that grammar, is not taken into account in GQE, at least not for now. More details about this option will be given in the next chapter.

5.2 AG ANALYSIS: LISP LANGUAGE

```
grammar Lisp;

lisp returns [int countN_out, int countW_out]
@init{ int countN_in = 0, countW_in = 0; }
  : sExp[countN_in, countW_in]
    {$countN_out = $sExp.countN_out;
     $countW_out = $sExp.countW_out;
     System.out.println("Total of numbers: " + $countN_out + "\n" + "Total of
         words: " + $countW_out + "\n");}
  ;

sExp[ int countN_in, int countW_in] returns [int countN_out, int countW_out]
  : NUM
    { $countN_out = $countN_in +1;
      $countW_out = $countW_in; }
  | WRD
    { $countN_out = $countN_in;
      $countW_out = $countW_in +1; }
  | '(' sExpList[countN_in, countW_in] ')'
    { $countN_out = $sExpList.countN_out;
      $countW_out = $sExpList.countW_out; }
  ;

sExpList[ int countN_in, int countW_in] returns [int countN_out, int countW_out]
@init{ int aux1, aux2; }
  : a1=sExp[countN_in, countW_in]
    { aux1 = $a1.countN_out;
      aux2 = $a1.countW_out; }
    a2=sExpList[aux1, aux2]
    { $countN_out = $a2.countN_out;
      $countW_out = $a2.countW_out; }
  |
    { $countN_out = $countN_in;
      $countW_out = $countW_in; }
  ;
```

Listing 5.2: A Attribute Grammar example for specifying Lisp language.

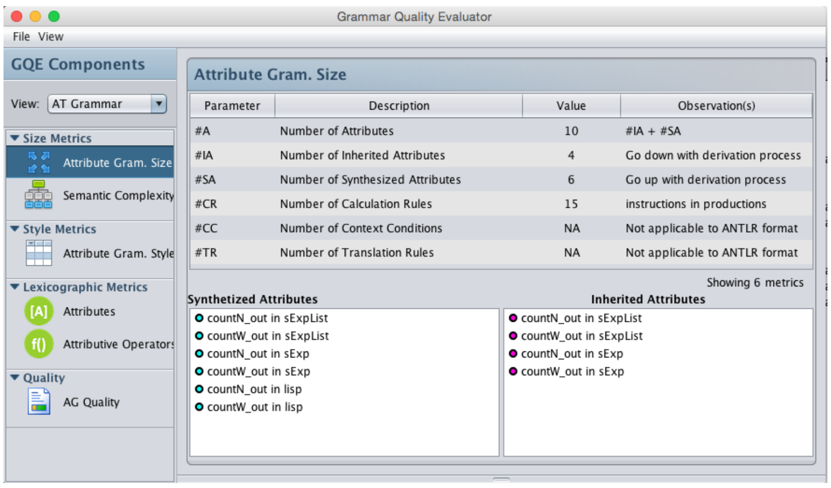
Following the same structure of the previous section, a new case study will be introduced and analyzed, but this time in order to an Attribute Grammar. Listing 5.2 exhibit the same context-free grammar for the *Lisp* language, exposed in Listing 5.1, extended with a semantic component defined at the expense of attributes. It is precisely this semantic component that GQE will automatically assess, thanks to the implementation of the algorithms, exposed in Subsection 4.3.2. Note that the quality assessment done previously stays valid for this grammar, as attribute grammars definition suggest.

In order to elucidate the speech taken hereinafter, the semantic component of this attribute grammar is responsible to calculate the total of number(*NUM*) and words(*WRD*) sequenced in the list[2]. This task is fulfilled by sending data, in this case counters, up and down between productions through synthesized and inherited attributes, and increment them when a number or a word is recognized.

– **Metrics Evaluation**

Figure 23 shows the results displayed by GQE for the set of attribute grammar size metrics defined in Table 15. Again, the results provided are in agreement with the desire, because exists:

- a total of 10 attributes $= (2+4+4)$;
- 4 inherited attributes $= (0+2+2)$;
- 6 synthesized attributes $= (2+2+2)$;
- 15 calculation rules $= (3+2+2+2+4+2)$;
- just as justified before, in the previous chapter, the number of Context Conditions and number of Translation Rules metrics are impossible to evaluate, for this ANTLR format.[3]



Parameter	Description	Value	Observation(s)
#A	Number of Attributes	10	#IA + #SA
#IA	Number of Inherited Attributes	4	Go down with derivation process
#SA	Number of Synthesized Attributes	6	Go up with derivation process
#CR	Number of Calculation Rules	15	instructions in productions
#CC	Number of Context Conditions	NA	Not applicable to ANTLR format
#TR	Number of Translation Rules	NA	Not applicable to ANTLR format

Showing 6 metrics

Synthetized Attributes
- countN_out in sExpList
- countW_out in sExpList
- countN_out in sExp
- countW_out in sExp
- countN_out in lisp
- countW_out in lisp

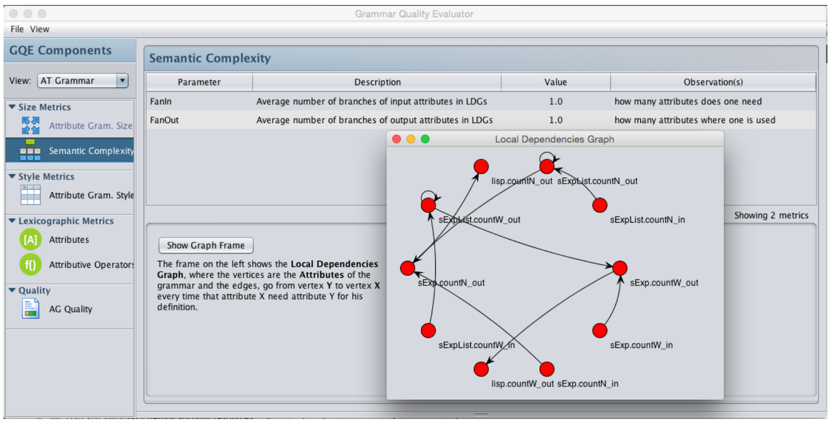
Inherited Attributes
- countN_out in sExpList
- countW_out in sExpList
- countN_out in sExp
- countW_out in sExp

Figure 23.: Computed grammar size metrics for AG specifying Lisp language.

The output for the pair of metrics used to assess the semantic complexity listed in Table 16 are exposed in Figure 24. Identically to what was shown for the syntax complexity metrics, here the GQE interface shows the Local Dependencies Graph, related to each attribute definition. Looking to the grammar and to the own graph, it is clearly that:

2 If the syntax of Attribute Grammars and its components is not clear to understand, please consult the Appendix B.
3 In the future, some others grammars formats can be implemented in GQE, thenceforth those and other metrics are able for evaluation.

- each attribute, on average, need one attribute for his definition - FanIn$= (1+0+0+2+0+2+1+2+2+0)/10$;
- each attribute, on average, is used in one attribute definition - FanOut$= (0+1+1+1+1+2+0+2+1+1)/10$;



Parameter	Description	Value	Observation(s)
FanIn	Average number of branches of input attributes in LDGs	1.0	how many attributes does one need
FanOut	Average number of branches of output attributes in LDGs	1.0	how many attributes where one is used

The frame on the left shows the **Local Dependencies Graph**, where the vertices are the **Attributes** of the grammar and the edges, go from vertex **Y** to vertex **X** every time that attribute X need attribute Y for his definition.

Figure 24.: Computed semantic complexity metrics for AG specifying Lisp language.

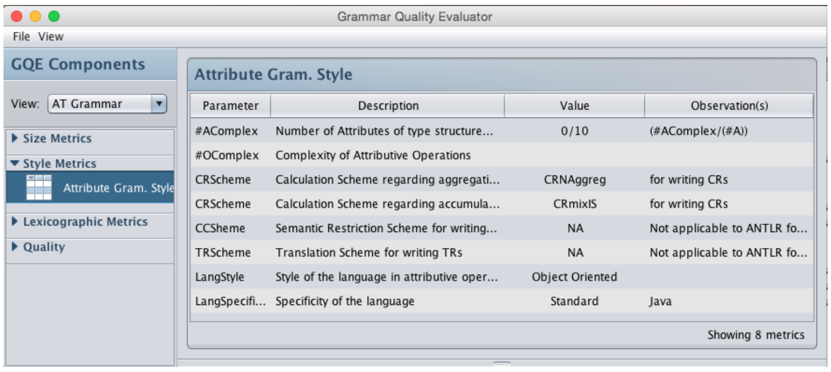
As for the results of the style metrics set, regarding now, attribute grammars and defined in Table 17, Table 18, Table 19, Table 20, Table 21 and Table 22, are presented in Figure 25. Once more, this results are correct, because:

- there is no attribute with a complex type - all attributes have a atomic type(*int*);
- the calculation rules follow the non aggregation pattern - all the partial values are maintained in simple distinct attributes - and a mixed pattern for the values accumulation - information is carried up and down between the productions;
- once more, for the same reason already explained, the Context Conditions Scheme and Translation Rules Scheme metrics are not evaluated;
- the style of the language is, in fact, object oriented;
- and the language specificity for writing attributive operations is Java and therefore Standard.

Figure 26 expose the results for the lexicographic metric, Clear Identifiers for Attributes, defined in Table 23. This set of metrics is completed with the metric listed in Table 24 related with Clear Identifiers for the Attributive Operator in calculation rules. Since this grammar example does not have any attributive operator to be assessed, the resultant panel displayed by GQE is not exposed here.

Taking into account, once more, the Definition 12 it is possible to check that all the identifiers used for attributes are clear:

- the identifier *countW_out* derives from the concept terms - *Count*, *Word* and *Out*;


Grammar Quality Evaluator

File View

GQE Components

View: AT Grammar

- Size Metrics
- Style Metrics
  - Attribute Gram. Style
- Lexicographic Metrics
- Quality

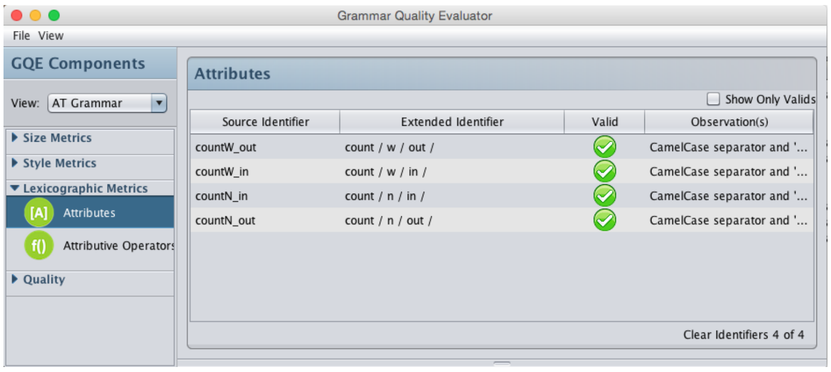
Attribute Gram. Style

Parameter	Description	Value	Observation(s)
#AComplex	Number of Attributes of type structure...	0/10	(#AComplex/(#A))
#OComplex	Complexity of Attributive Operations		
CRScheme	Calculation Scheme regarding aggregati...	CRNAggreg	for writing CRs
CRScheme	Calculation Scheme regarding accumula...	CRmixIS	for writing CRs
CCSheme	Semantic Restriction Scheme for writing...	NA	Not applicable to ANTLR fo...
TRScheme	Translation Scheme for writing TRs	NA	Not applicable to ANTLR fo...
LangStyle	Style of the language in attributive oper...	Object Oriented	
LangSpecifi...	Specificity of the language	Standard	Java

Showing 8 metrics

Figure 25.: Computed style metrics for AG specifying Lisp language.

- the identifier *countW_in* derives from the concept terms - *Count*, *Word* and *In*;
- the identifier *countN_in* derives from the concept terms - *Count*, *Number* and *In*;
- and the identifier *countN_out* derives from the concept terms - *Count*, *Number* and *Out*.


Grammar Quality Evaluator

File View

GQE Components

View: AT Grammar

- Size Metrics
- Style Metrics
- Lexicographic Metrics
  - [A] Attributes
  - f() Attributive Operators
- Quality

Attributes

☐ Show Only Valids

Source Identifier	Extended Identifier	Valid	Observation(s)
countW_out	count / w / out /	✓	CamelCase separator and '...
countW_in	count / w / in /	✓	CamelCase separator and '...
countN_in	count / n / in /	✓	CamelCase separator and '...
countN_out	count / n / out /	✓	CamelCase separator and '...

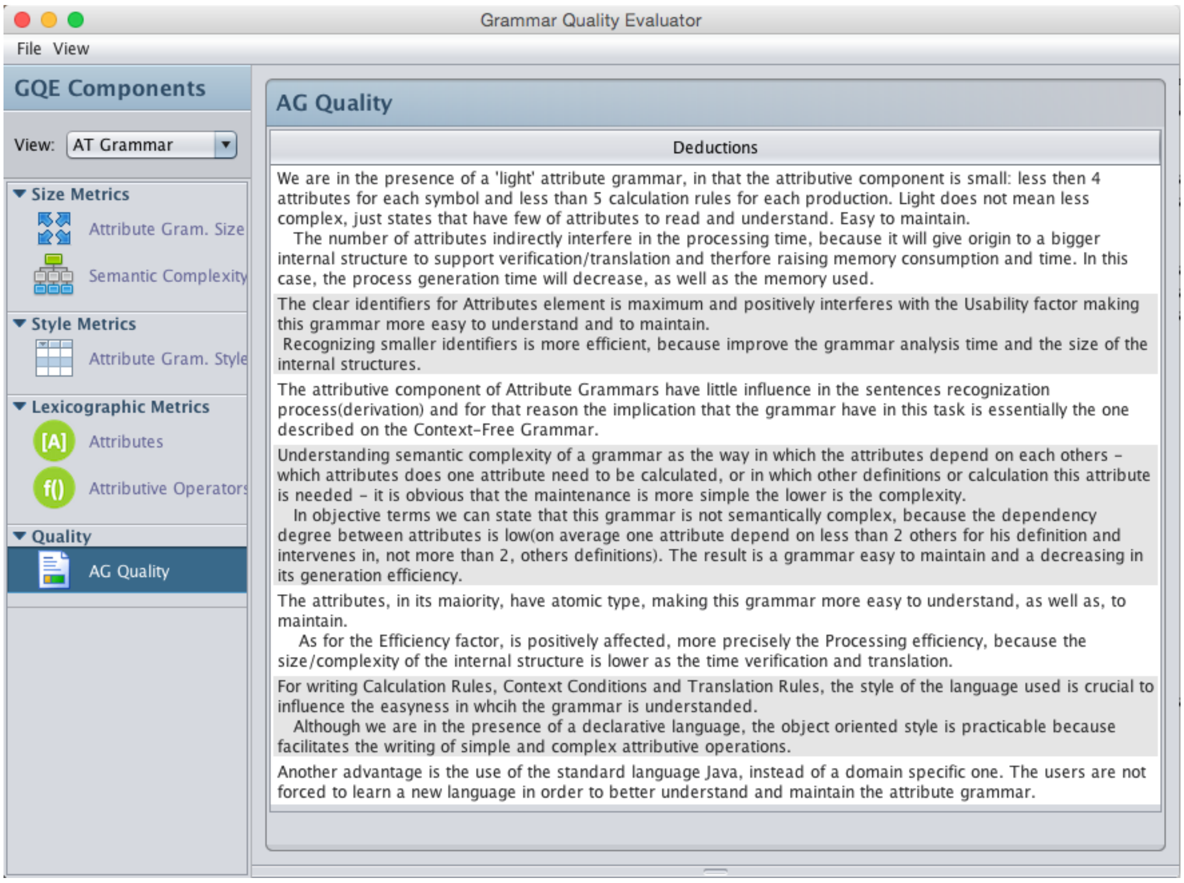
Clear Identifiers 4 of 4

Figure 26.: Computed lexicographic metrics for AG specifying Lisp language.

- **Assessment Report**

  After all the previous set of metrics being evaluated, and similarly to what was done for context-free grammar, a quality assessment report is displayed by GQE, as shown in Figure 27. By crossing the metric values with their impact on the quality of the grammar - listed in Table 3 - the tool generate a set of quality assumptions about the attribute grammar in study.

  By analyzing all the assumptions provided and looking for the grammar in question, it is easy to verify that all of them make sense and states about some quality factor.



AG Quality

Deductions

We are in the presence of a 'light' attribute grammar, in that the attributive component is small: less then 4 attributes for each symbol and less than 5 calculation rules for each production. Light does not mean less complex, just states that have few of attributes to read and understand. Easy to maintain.
The number of attributes indirectly interfere in the processing time, because it will give origin to a bigger internal structure to support verification/translation and therfore raising memory consumption and time. In this case, the process generation time will decrease, as well as the memory used.

The clear identifiers for Attributes element is maximum and positively interferes with the Usability factor making this grammar more easy to understand and to maintain.
Recognizing smaller identifiers is more efficient, because improve the grammar analysis time and the size of the internal structures.

The attributive component of Attribute Grammars have little influence in the sentences recognization process(derivation) and for that reason the implication that the grammar have in this task is essentially the one described on the Context-Free Grammar.

Understanding semantic complexity of a grammar as the way in which the attributes depend on each others – which attributes does one attribute need to be calculated, or in which other definitions or calculation this attribute is needed – it is obvious that the maintenance is more simple the lower is the complexity.
In objective terms we can state that this grammar is not semantically complex, because the dependency degree between attributes is low(on average one attribute depend on less than 2 others for his definition and intervenes in, not more than 2, others definitions). The result is a grammar easy to maintain and a decreasing in its generation efficiency.

The attributes, in its maiority, have atomic type, making this grammar more easy to understand, as well as, to maintain.
As for the Efficiency factor, is positively affected, more precisely the Processing efficiency, because the size/complexity of the internal structure is lower as the time verification and translation.

For writing Calculation Rules, Context Conditions and Translation Rules, the style of the language used is crucial to influence the easyness in whcih the grammar is understanded.
Although we are in the presence of a declarative language, the object oriented style is practicable because facilitates the writing of simple and complex attributive operations.

Another advantage is the use of the standard language Java, instead of a domain specific one. The users are not forced to learn a new language in order to better understand and maintain the attribute grammar.

Figure 27.: Assessment report for AG specifying Lisp language.

6 CONCLUSIONS

6.1 CONCLUSIONS

This final chapter is a summary, or a synopsis, of everything that was exposed throughout this document, remembering the aims of the project and how they were achieved. Just as stated in the Introduction, a critical reflection will be done in this chapter, discussing some aspects of the research and development work done, as well as what is expected for future work.

First, the document starts to contextualize the grammars in the software engineering, trough all software products that as the core knowledge based on grammar, the defined grammarware and how the aspects of engineering, such as the quality, are not being used in its production. Then is presented the motivation for this masters' work. So, to assess the quality of any software application several techniques and methods can be applied, but in this case regarding the grammars, one is introduced - Measurement.

It was discussed that for applying the measurement theory to any object, is necessary to identify the attributes to be measured, the entities in which such attributes are measured and how is measured. The theory behind measurement say that for measuring some attribute, like quality on software products, which is an attribute not directly measurable, we need to first characterize the attribute for a certain entity. After that, this document discusses the assessment of attribute grammars' quality.

Following the previous reasoning, first the quality characteristics for Languages were identified, because it is also a challenge of this thesis, relating the quality of a grammar with the quality of the language generated by that grammar. To assess the language quality, eight characteristics were defined: Expressiveness, Documentation, Unicity, Consistency, Extensability, Scalability, Reliability and Modularity. The more characteristics a language holds the better it is. For each characteristic, was measured the influence over the four critical factors to quality of the language: learning, writing, understanding and the efficiency in recognition.

Then, the same procedure was applied to grammar, more precisely, Context-Free Grammars and Attribute Grammars. The quality of the Grammar was defined as, while specification that generates the language, in terms of usability, and while specification that generates a processor, in terms of efficiency. The influence that some grammar elements have over this quality factors was also discussed.

As result of all the reasoning made, a set of metrics is introduced to assess the quality of Context-Free Grammars and Attribute Grammars. This set is divided in 3 subsets: the common Size, Style/-Form metrics, and the innovative lexicographic metrics, considered useful for retaining information related with the quality of the generated language.

Assuming the grammars as the instruments of this work and converging all the theory until then dissected arises then the *GQE* - **Grammar Quality Evaluator** system. GQE is a tool for assessing automatically the quality of Context-Free Grammars and Attribute Grammars, at the expense of the metrics evaluation. The first concern was to build the data structures needed for applying on them algorithms that allowed a correct implementation of each introduced metric. Then a reasoning mechanism was implemented, responsible to read the values of each metric and display an assessment report about the grammar in question.

By analyzing the quality reports displayed by GQE, it is concluded that, in fact, is possible to assess the quality of a grammar automatically, at the expense of the metrics introduced here, in terms of its *usability* (*understanding, derivation and maintenance* as a tool for generating some language and *efficiency* as a tool for generating the parser for that language.

Remembering the aims proposed for this masters work and checking the accomplished achievements, it is possible to infer that:

1. A grammar-based software tool for software engineers and others developers in the language processing field, was developed accordingly the desired requirements:
   - a new set of grammar metrics were introduced in accordance with the quality characteristics present both in grammars and in the languages generated by those grammars;
   - those metrics were implemented successfully in the system, allowing the attributes evaluation;
   - accept a given grammar (CFG or AG) and assess their quality;
   - manipulating a grammar and transforming it into another equivalent grammar with better quality is not automatically done by GQE. Although, it is clear to see that this process of increasing the quality of the grammar by transformation can be done easily now. In fact, implementing this feature into this tool it will be very laborious (in terms of coding and interfacing), when with the quality results the grammar developer can do it fast and without effort.
2. Well-founded and proven methods and techniques were provided to support the development, maintenance, recovery and implementation of attribute grammars. Through the application of a scientific method called *measurement* and with the respective measurement techniques discussed in the State of the Art chapter, attribute grammar processes can be aided.
3. Better quality grammar-based software systems can be achieved with the aid of GQE, because the development process of context-free grammars and attribute grammars can be supervised

and controlled easily. Therefore and by systematically apply this solution and reasoning about the quality assessment that this provide, there is room also for improving critical aspects such as performance, reliability and efficiency in a sustained way.

6.2 FUTURE WORK

It is intended to discuss here some circumstances about the thread of possible future investigations following this dissertation. It is noted, obvious, that it still exist a lot of work to be done in this field due to the relevance of the subject and the advantages that future projects can bring to software engineering.

- Relatively to the Grammar Quality Evaluator there is room for improvement. More grammar formats can be added to the range of this tool, implementing then some of the metrics that were proven to be impossible to evaluate for the ANTLR format. Also, taking this system to next level, by providing some artificial intelligence techniques regarding the quality of the already assessed grammars.

  The idea is to create a base of knowledge for grammars and therefore improving the specificity of the quality assumptions displayed. This can be achieved by saving the opinions of the grammar engineers users and in consequence teaching automatically the system what could be good or poor grammar in terms of quality.

  Although the obvious involved complexity, it would be nice to have some sort of stable *formula*, calculated using the values of the assessed metrics, that gives us some final verdict about the quality of the grammars. For sure, that is the dream!

- After being proven, by the content of this dissertation, that from the quality assessment made on a grammar it is possible to draw conclusions about the quality generated by that grammar, the intention is to do it automatically. This can be accomplished by giving an intensive studying upon the subject languages quality, just as the one made here for grammars, and relating that with the product of this dissertation;

- And to finalize, continue gifting the grammar engineering and the languages processing field with more established and well-founded methods of engineering.

BIBLIOGRAPHY

Tiago L. Alves and Joost Visser. Metrication of sdf grammars. Technical report, Universidade do Minho, May 2005. URL `http://wiki.di.uminho.pt/twiki/pub/Personal/Tiago/Publications/DI-PURe-05-05-01.pdf`.

Tiago L. Alves and Joost Visser. A case study in grammar engineering. In *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, pages 285–304, 2008. doi: 10.1007/978-3-642-00434-6_18. URL `http://dx.doi.org/10.1007/978-3-642-00434-6_18`.

Emily M. Bender. 1 grammar engineering for linguistic hypothesis testing, April 2007. URL `http://faculty.washington.edu/ebender/papers/TLSX_preprint.pdf`.

Barry W Boehm, John R Brown, Hans Kaspar, and Myron Lipow. *Characteristics of software quality*. TRW Softw. Technol. North-Holland, Amsterdam, 1978.

Nuno Ramos Carvalho, José João Almeida, Pedro Rangel Henriques, and Maria João Varanda Pereira. From source code identifiers to natural language terms. *Journal of Systems and Software*, 100:117–128, 2015. doi: 10.1016/j.jss.2014.10.013. URL `http://dx.doi.org/10.1016/j.jss.2014.10.013`.

Joseph P. Cavano and James A. McCall. A framework for the measurement of software quality. *SIGSOFT Softw. Eng. Notes*, 3(5):133–139, January 1978. ISSN 0163-5948. doi: 10.1145/953579.811113. URL `http://doi.acm.org/10.1145/953579.811113`.

Matej Crepinsek, Tomaz Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, and Gilles Roussel. On automata and language based grammar metrics. *Comput. Sci. Inf. Syst.*, 7(2):309–329, 2010. doi: 10.2298/CSIS1002309C. URL `http://dx.doi.org/10.2298/CSIS1002309C`.

Jelle de Groot, Ariadi Nugroho, Thomas Bäck, and Joost Visser. What is the value of your software? In *Proceedings of the Third International Workshop on Managing Technical Debt, MTD 2012, Zurich, Switzerland, June 5, 2012*, pages 37–44, 2012. URL `http://dl.acm.org/citation.cfm?id=2666043`.

Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer, 1988. ISBN 3-540-50056-1. doi: 10.1007/BFb0030509. URL `http://dx.doi.org/10.1007/BFb0030509`.

Gregor Erbach. Tools for grammar engineering. In *Proceedings of the Third Conference on Applied Natural Language Processing*, ANLC '92, pages 243–244, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics. doi: 10.3115/974499.974548. URL http://dx.doi.org/10.3115/974499.974548.

Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998. ISBN 0534954251.

A. A Ghani and R. B. Hunter. An attribute grammar approach to specifying halstead's metrics. *Malaysian Journal of Computer Science*, 9, 1996.

Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings*, pages 30–39, 2007. doi: 10.1109/QUATIC.2007.8. URL http://dx.doi.org/10.1109/QUATIC.2007.8.

Pedro Rangel Henriques. Brincando às Linguagens com Rigor: Engenharia Gramatical. (habilitation in cs) technical report, CCTC/DI, Univeristy of Minho, November 2013.

C. A. R. Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, Stanford, CA, USA, 1973.

James W. Howatt. A project-based approach to programming language evaluation. *SIGPLAN Notices*, 30(7):37–40, 1995. doi: 10.1145/208639.208642. URL http://doi.acm.org/10.1145/208639.208642.

M Jørgensen. Software quality measurement. *Advances in Engineering Software*, 30(12):907 – 912, 1999. ISSN 0965-9978. doi: http://dx.doi.org/10.1016/S0965-9978(99)00015-0. URL http://www.sciencedirect.com/science/article/pii/S0965997899000150.

Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201729156.

Chris F Kemerer. An empirical validation of software cost estimation models. *Commun. ACM*, 30 (5):416–429, May 1987. ISSN 0001-0782. doi: 10.1145/22899.22906. URL http://doi.acm.org/10.1145/22899.22906.

Barbara A. Kitchenham. *Software Metrics: Measurement for Software Process Improvement*. Blackwell Publishers, Inc., Cambridge, MA, USA, 1996. ISBN 1855548208.

Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, July 2005. ISSN 1049-331X. doi: 10.1145/1072997.1073000. URL http://doi.acm.org/10.1145/1072997.1073000.

Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. doi: 10.1007/BF01692511. URL http://dx.doi.org/10.1007/BF01692511.

Ralf Lämmel. Grammar testing. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 201–216, 2001. doi: 10.1007/3-540-45314-8_15. URL http://dx.doi.org/10.1007/3-540-45314-8_15.

Robert E Park, Wolfhart B Goethert, and William A Florac. Goal-driven software measurement. a guidebook. Technical report, DTIC Document, 1996.

Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013. ISBN 1934356999, 9781934356999.

James F. Power and Brian A. Malloy. Metric-based analysis of context-free grammars. In *8th International Workshop on Program Comprehension (IWPC 2000), 10-11 June 2000, Limerick, Ireland*, pages 171–178, 2000. doi: 10.1109/WPC.2000.852491. URL http://dx.doi.org/10.1109/WPC.2000.852491.

James F. Power and Brian A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance*, 16(6):405–426, 2004. doi: 10.1002/smr.293. URL http://dx.doi.org/10.1002/smr.293.

Robert W. Sebesta. *Concepts of programming languages (9. ed.)*. Addison-Wesley-Longman, 2009. ISBN 978-0-201-38596-0.

Dan Shoemaker and Nancy R Mead. Software assurance measurement–state of the practice. Technical report, Software Engineering Institute, Nov 2013.

David A. Watt and William Findlay. *Programming language design concepts*. Wiley, 2004. ISBN 978-0-470-85320-7.

A

GRAMMAR OF ANTLR META-LANGUAGE

To give the reader a little notion of what is behind the development of GQE, this appendix shows the complete Context-Free Grammar to validate ANTLR meta-language.

The initial grammar was developed by Sam Harwell and Terrence Parr, but was extended to this grammar by the author with the intention of including the Java language specification, valid in some parts of the ANTLR meta-language such as actions, attributes declarations and others. It is important to point out that only the parser grammar is shown because it is enough to understand how the language is recognized.

```
parser grammar ANTLRv4Parser;

options {
    tokenVocab=ANTLRv4Lexer;
}

// The main entry point for parsing a v4 ANTLR Grammar.
grammarSpec
    :   (DOC_COMMENT)?
        grammarType id SEMI
        prequelConstruct*
        rules
        modeSpec*
        EOF
    ;

grammarType
    :   (   LEXER GRAMMAR
        |   PARSER GRAMMAR
        |   GRAMMAR
        )
    ;

// This is the list of all constructs that can be declared before
// the set of rules that compose the grammar, and is invoked 0..n
// times by the grammarPrequel rule.
prequelConstruct
```

```
        :   optionsSpec
        |   delegateGrammars
        |   tokensSpec
        |   action
        ;

// A list of options that affect analysis and/or code generation
optionsSpec
        :   OPTIONS (option SEMI)* RBRACE
        ;

option
        :   id ASSIGN optionValue
        ;

optionValue
        :   id (DOT id)*
        |   STRING_LITERAL
        |   ACTION
        |   INT
        ;

delegateGrammars
        :   IMPORT delegateGrammar (COMMA delegateGrammar)* SEMI
        ;

delegateGrammar
        :   id ASSIGN id
        |   id
        ;

tokensSpec
        :   TOKENS id (COMMA id)* COMMA? RBRACE
        ;

/** Match stuff like @parser::members {int i;} */
action
        :   AT (actionScopeName COLONCOLON)? id LBRACE compilationUnit RBRACE
        ;

/** Sometimes the scope names will collide with keywords; allow them as
 *  ids for action scopes.
 */
actionScopeName
        :   id
        |   LEXER
        |   PARSER
	;

modeSpec
	:	MODE id SEMI lexerRule*
	;

rules
	:	ruleSpec*
	;

ruleSpec
	:	parserRuleSpec
	|	lexerRule
	;

parserRuleSpec
	:	DOC_COMMENT?
		ruleModifiers?
		attrParameters[0]?
		ruleReturns? throwsSpec? localsSpec?
		rulePrequel*
		COLON r=ruleBlock SEMI
		exceptionGroup
	;

// Rules for parsing JAVA code inside attributes, actions, members, etc ..
compilationUnit
	:	packageDeclaration? importDeclaration* typeDeclaration*
	;

packageDeclaration
	:	annotation* PACKAGE qualifiedName SEMI
	;

importDeclaration
	:	IMPORT STATIC? qualifiedName (DOT STAR)? SEMI
	;

typeDeclaration
	:	classOrInterfaceModifier* classDeclaration
	|	classOrInterfaceModifier* enumDeclaration
	|	classOrInterfaceModifier* interfaceDeclaration
	|	classOrInterfaceModifier* annotationTypeDeclaration
	|	SEMI
	;

modifier
```

```
        :   classOrInterfaceModifier
        |   (   NATIVE
            |   SYNC
            |   TRANSIENT
            |   VOLATILE
            )
        ;

classOrInterfaceModifier
        :   annotation          // class or interface
        |   (   PUBLIC      // class or interface
            |   PROTECTED   // class or interface
            |   PRIVATE     // class or interface
            |   STATIC      // class or interface
            |   ABSTRACT    // class or interface
            |   FINAL      // class only -- does not apply to interfaces
            |   STRICTFP    // class or interface
            )
        ;

variableModifier
        :   FINAL
        |   annotation
        ;

classDeclaration
        :   CLASS id typeParameters?
            (EXTENDS type)?
            (IMPLEMENTS typeList)?
            classBody
        ;

typeParameters
        :   LT typeParameter (COMMA typeParameter)* GT
        ;

typeParameter
        :   id (EXTENDS typeBound)?
        ;

typeBound
        :   type (AND type)*
        ;

enumDeclaration
        :   ENUM id (IMPLEMENTS typeList)?
            LBRACE enumConstants? COMMA? enumBodyDeclarations? RBRACE
        ;

enumConstants
        :    enumConstant  (COMMA  enumConstant)*
        ;

enumConstant
        :    annotation*  id  arguments?  classBody?
        ;

enumBodyDeclarations
        :    SEMI  classBodyDeclaration*
        ;

interfaceDeclaration
        :    INTERFACE  id  typeParameters?  (EXTENDS  typeList)?  interfaceBody
        ;

classBody
        :    LBRACE  classBodyDeclaration*  RBRACE
        ;

interfaceBody
        :    LBRACE  interfaceBodyDeclaration*  RBRACE
        ;

classBodyDeclaration
        :    SEMI
        |    STATIC?  blockS
        |    modifier*  memberDeclaration
        ;

memberDeclaration
        :    methodDeclaration
        |    genericMethodDeclaration
        |    fieldDeclaration
        |    constructorDeclaration
        |    genericConstructorDeclaration
        |    interfaceDeclaration
        |    annotationTypeDeclaration
        |    classDeclaration
        |    enumDeclaration
        ;

/* We use rule this even for void methods which cannot have [] after parameters.
   This simplifies grammar and we can consider void to be a type, which
   renders the [] matching as a context-sensitive issue or a semantic check
   for invalid return type after parsing.
  */
methodDeclaration
    :   (type|VOID) id formalParameters (LBRAC RBRAC)*
        (THROWS qualifiedNameList)?
        (   methodBody
        |   SEMI
        )
    ;

genericMethodDeclaration
    :   typeParameters methodDeclaration
    ;

constructorDeclaration
    :   id formalParameters (THROWS qualifiedNameList)?
        constructorBody
    ;

genericConstructorDeclaration
    :   typeParameters constructorDeclaration
    ;

fieldDeclaration
    :   type variableDeclarators SEMI
    ;

interfaceBodyDeclaration
    :   modifier* interfaceMemberDeclaration
    |   SEMI
    ;

interfaceMemberDeclaration
    :   constDeclaration
    |   interfaceMethodDeclaration
    |   genericInterfaceMethodDeclaration
    |   interfaceDeclaration
    |   annotationTypeDeclaration
    |   classDeclaration
    |   enumDeclaration
    ;

constDeclaration
    :   type constantDeclarator (COMMA constantDeclarator)* SEMI
    ;

constantDeclarator
```

```
        :   id  (LBRAC RBRAC)* ASSIGN variableInitializer
        ;

// see matching of [] comment in methodDeclaratorRest
interfaceMethodDeclaration
        :   (type|VOID) id formalParameters (LBRAC RBRAC)*
            (THROWS qualifiedNameList)?
            SEMI
        ;

genericInterfaceMethodDeclaration
        :   typeParameters interfaceMethodDeclaration
        ;

variableDeclarators
        :   variableDeclarator (COMMA variableDeclarator)*
        ;

variableDeclarator
        :   variableDeclaratorId (ASSIGN variableInitializer)?
        ;

qualifiedNameList
        :   qualifiedName (COMMA qualifiedName)*
        ;

formalParameters
        :   LPAREN formalParameterList? RPAREN
        ;

formalParameterList
        :   formalParameter (COMMA formalParameter)* (COMMA lastFormalParameter)?
        |   lastFormalParameter
        ;

formalParameter
        :   variableModifier* type variableDeclaratorId
        ;

lastFormalParameter
        :   variableModifier* type SUSP_POINTS variableDeclaratorId
        ;

methodBody
        :   blockS
        ;
```

```
constructorBody
        :   blockS
        ;

qualifiedName
        :   id  (DOT  id)*
        ;

// Match annotations
annotation
        :   AT  annotationName  ( LPAREN  ( elementValuePairs  |  elementValue  )?  RPAREN  )
            ?
        ;

annotationName : qualifiedName ;

elementValuePairs
        :   elementValuePair  (COMMA  elementValuePair)*
        ;

elementValuePair
        :   id  ASSIGN  elementValue
        ;

elementValue
        :   expression
        |   annotation
        |   elementValueArrayInitializer
        ;

elementValueArrayInitializer
        :   RBRACE  (elementValue  (COMMA  elementValue)*)?  (COMMA)?  RBRACE
        ;

annotationTypeDeclaration
        :   AT  INTERFACE  id  annotationTypeBody
        ;

annotationTypeBody
        :   LBRACE  (annotationTypeElementDeclaration)*  RBRACE
        ;

annotationTypeElementDeclaration
        :   modifier*  annotationTypeElementRest
        |   SEMI  // this is not allowed by the grammar, but apparently allowed by the
             actual compiler
        ;
```

```
annotationTypeElementRest
        :     type  annotationMethodOrConstantRest  SEMI
        |     classDeclaration  SEMI?
        |     interfaceDeclaration  SEMI?
        |     enumDeclaration  SEMI?
        |     annotationTypeDeclaration  SEMI?
        ;

annotationMethodOrConstantRest
        :     annotationMethodRest
        |     annotationConstantRest
        ;

annotationMethodRest
        :     id  LPAREN  RPAREN  defaultValue?
        ;

annotationConstantRest
        :     variableDeclarators
        ;

defaultValue
        :     DEFAULT  elementValue
        ;

// Match Java statements and blocks

blockS
        :     LBRACE  blockStatement*  RBRACE
        ;

blockStatement
        :     localVariableDeclarationStatement
        |     statement
        |     typeDeclaration
        ;

localVariableDeclarationStatement
        :      localVariableDeclaration  SEMI
        ;

localVariableDeclaration
        :     variableModifier*  type  variableDeclarators
        ;

statement
```

```
    :   blockS
    |   ASSERT expression (COLON expression)? SEMI
    |   IF parExpression statement (ELSE statement)?
    |   FOR LPAREN forControl RPAREN statement
    |   WHILE parExpression statement
    |   DO statement WHILE parExpression SEMI
    |   TRY blockS (catchClause+ finallyBlock? | finallyBlock)
    |   TRY resourceSpecification blockS catchClause* finallyBlock?
    |   SWITCH parExpression LBRACE switchBlockStatementGroup* switchLabel*
       RBRACE
    |   SYNC parExpression blockS
    |   RETURN expression? SEMI
    |   THROW expression SEMI
    |   BREAK id? SEMI
    |   CONTINUE id? SEMI
    |   SEMI
    |   statementExpression SEMI
    |   id COLON statement
    ;

catchClause
    :   CATCH LPAREN variableModifier* catchType id RPAREN blockS
    ;

catchType
    :   qualifiedName (OR qualifiedName)*
    ;

finallyBlock
    :   FINALLY blockS
    ;

resourceSpecification
    :   LPAREN resources SEMI? RPAREN
    ;

resources
    :   resource (SEMI resource)*
    ;

resource
    :   variableModifier* classOrInterfaceType variableDeclaratorId ASSIGN
       expression
    ;

// Match the ANTLR attributes
attrParameters
```

```
            : LBRAC attrList RBRAC
            ;

attrList
            : attrParameter (COMMA attrParameter)*
            ;

attrParameter
            : type variableDeclaratorId (ASSIGN variableInitializer)?
            ;

variableInitializer
        :   arrayInitializer
        |   expression
        ;

arrayInitializer
        :   LBRACE (variableInitializer (COMMA variableInitializer)* (COMMA)? )?
           RBRACE
        ;
/** Matches cases then statements, both of which are mandatory.
 *  To handle empty cases at the end, we add switchLabel* to statement.
 */
switchBlockStatementGroup
        :   switchLabel+ blockStatement+
        ;

switchLabel
        :   CASE constantExpression COLON
        |   CASE enumConstantName COLON
        |   DEFAULT COLON
        ;

enumConstantName
        :   id
        ;

forControl
        :   enhancedForControl
        |   forInit? SEMI expression? SEMI forUpdate?
        ;

forInit
        :   localVariableDeclaration
        |   expressionList
        ;
```

```
enhancedForControl
        :    variableModifier∗ type variableDeclaratorId COLON expression
        ;

forUpdate
        :    expressionList
        ;

// Match all kind of Java expressions
parExpression
        :    LPAREN expression RPAREN
        ;

statementExpression
        :    expression
        ;

constantExpression
        :    expression
        ;

expressionList
        :    expression (COMMA expression)∗
        ;

expression
        :    primary
        |    expression DOT id
        |    expression DOT THIS
        |    expression DOT NEW nonWildcardTypeArguments? innerCreator
        |    expression DOT SUPER superSuffix
        |    expression DOT explicitGenericInvocation
        |    expression LBRAC expression RBRAC
        |    expression LPAREN (expressionList)? RPAREN
        |    NEW creator
        |    LPAREN type RPAREN expression
        |    expression (DOUBLE_PLUS | DOUBLE_MINUS)
        |    (PLUS|MINUS|DOUBLE_PLUS|DOUBLE_MINUS) expression
        |    (NOT|EXCL) expression
        |    expression (STAR|DIV|PERCENT) expression
        |    expression (PLUS|MINUS) e2=expression
        |    expression (LT LT | GT GT GT | GT GT) expression
        |    expression (LTE | GTE | GT | LT) expression
        |    expression INSTANCEOF type
        |    expression (EQUAL | NOT_EQUAL) expression
        |    expression AND expression
        |    expression EXP expression
        |     expression OR expression
        |     expression DOUBLE_AND expression
        |     expression DOUBLE_OR expression
        |     expression QUESTION expression COLON expression
        |     expression
              (     ASSIGN
              |     PLUS_ASSIGN
              |     MINUS_ASSIGN
              |     STAR_ASSIGN
              |     DIV_ASSIGN
              |     AND_ASSIGN
              |     OR_ASSIGN
              |     EXP_ASSIGN
              |     GG_ASSIGN
              |     GGG_ASSIGN
              |     LL_ASSIGN
              |     PERCENT_ASSIGN
              )
              expression
        ;

primary
        :     LPAREN expression RPAREN
        |     THIS
        |     SUPER
        |     literal
        |     id
        |     DOLLAR id              // Acessing rule attributes
        |     type DOT CLASS
        |     VOID DOT CLASS
        |     nonWildcardTypeArguments (explicitGenericInvocationSuffix | THIS
             arguments)
        ;

explicitGenericInvocationSuffix
        :     SUPER superSuffix
        |     id arguments
        ;

literal
        :     IntegerLiteral
        |     FloatingPointLiteral
        |     BooleanLiteral
        |     NULL
        |     STRING_LITERAL
        ;
```

```

nonWildcardTypeArguments
    :   LT typeList GT
    ;

typeList
    :   type (COMMA type)*
    ;

creator
    :   nonWildcardTypeArguments createdName classCreatorRest
    |   createdName (arrayCreatorRest | classCreatorRest)
    ;

createdName
    :   id typeArgumentsOrDiamond? (DOT id typeArgumentsOrDiamond?)*
    |   primitiveType
    ;

typeArgumentsOrDiamond
    :   LT GT
    |   typeArguments
    ;

classCreatorRest
    :   arguments
    ;

innerCreator
    :   id nonWildcardTypeArgumentsOrDiamond? classCreatorRest
    ;

nonWildcardTypeArgumentsOrDiamond
    :   LT GT
    |   nonWildcardTypeArguments
    ;

superSuffix
    :   arguments
    |   DOT id arguments?
    ;

explicitGenericInvocation
    :   nonWildcardTypeArguments explicitGenericInvocationSuffix
    ;

arrayCreatorRest
```

```
        :    LBRAC
             (    RBRAC (LBRAC RBRAC)* arrayInitializer
             |    expression RBRAC (LBRAC expression RBRAC)* (LBRAC RBRAC)*
             )
        ;

arguments
        :    LPAREN expressionList? RPAREN
        ;

type
        :    classOrInterfaceType (LBRAC RBRAC)*
        |    primitiveType (LBRAC RBRAC)*
        ;

classOrInterfaceType
        :    id typeArguments? (DOT id typeArguments? )*
        ;

typeArguments
        :    LT typeArgument (COMMA typeArgument)* GT
        ;

typeArgument
        :    type
        |    QUESTION ((EXTENDS | SUPER) type)?
        ;

primitiveType
        :    BOOLEAN
        |    CHAR
        |    BYTE
        |    SHORT
        |    TINT
        |    LONG
        |    FLOAT
        |    DOUBLE
        ;

variableDeclaratorId
        :    id (LBRAC RBRAC)*
        ;

// End parsing Java code attributes

exceptionGroup
        :    exceptionHandler* finallyClause?
        ;

exceptionHandler
        :   CATCH attrParameters[2] LBRACE blockStatement* RBRACE
        ;

finallyClause
        :   FINALLY LBRACE blockStatement* RBRACE
        ;

rulePrequel
        :   optionsSpec
        |   ruleAction
        ;

ruleReturns
        :   RETURNS attrParameters[1]
        ;

throwsSpec
        :   THROWS id (COMMA id)*
        ;

localsSpec
        :   LOCALS attrParameters[3]
        ;

/** Match stuff like @init {int i;} */
ruleAction
        :   AT id LBRACE blockStatement* RBRACE
        ;

ruleModifiers
        :   ruleModifier+
        ;

ruleModifier
        :   PUBLIC
        |   PRIVATE
        |   PROTECTED
        |   FRAGMENT
        ;

ruleBlock
        :   ruleAltList
        ;

```

```
ruleAltList
    :   labeledAlt (OR labeledAlt)*
    ;

labeledAlt
    :   alternative (POUND id)?
    ;

lexerRule
    :   DOC_COMMENT? FRAGMENT?
        TOKEN_REF COLON lexerRuleBlock SEMI
    ;

lexerRuleBlock
    :   lexerAltList
    ;

lexerAltList
    :   lexerAlt (OR lexerAlt)*
    ;

lexerAlt
    :   lexerElements lexerCommands?
    |
    ;

lexerElements
    :   lexerElement+
    ;

lexerElement
    :   labeledLexerElement ebnfSuffix?
    |   lexerAtom ebnfSuffix?
    |   lexerBlock ebnfSuffix?
    |   LBRACE blockStatement* RBRACE QUESTION?
    ;

labeledLexerElement
    :   id (ASSIGN|PLUS_ASSIGN)
        (   lexerAtom
        |   block
        )
    ;

lexerBlock
    :   LPAREN lexerAltList RPAREN
    ;
```

```

// E.g., channel(HIDDEN), skip, more, mode(INSIDE), push(INSIDE), pop
lexerCommands
        :     RARROW lexerCommand (COMMA lexerCommand)*
        ;

lexerCommand
        :     lexerCommandName LPAREN lexerCommandExpr RPAREN
        |     lexerCommandName
        ;

lexerCommandName
        :     id
        |     MODE
        ;

lexerCommandExpr
        :     id
        |     INT
        ;

altList
        :     alternative (OR alternative)*
        ;

alternative
        :     elementOptions? (element)*
        ;

element
        :     labeledElement
              (     ebnfSuffix
              |
              )
        |     atom{
              (     ebnfSuffix
              |
              )
        |     ebnf
        |     LBRACE (blockStatement)* RBRACE (QUESTION)?
        ;

labeledElement
        :     id (ASSIGN|PLUS_ASSIGN)
              (     atom
              |     block
              )
        ;

ebnf
        :     block  blockSuffix?
        ;


blockSuffix
        :     ebnfSuffix
        ;

ebnfSuffix
        :     QUESTION  QUESTION?
        |     STAR  QUESTION?
        |     PLUS  QUESTION?
        ;

lexerAtom
        :     range
        |     terminal
        |     RULE_REF
        |     notSet
        |     LEXER_CHAR_SET
        |     DOT  elementOptions?
        ;

atom
        :     range // Range x..y - only valid in lexers
        |     terminal
        |     ruleref
        |     notSet
        |     DOT  elementOptions?
        ;

notSet
        :     NOT  setElement
        |     NOT  blockSet
        ;

blockSet
        :     LPAREN  setElement  (OR  setElement)*  RPAREN
        ;

setElement
        :     TOKEN_REF  elementOptions?
        |     STRING_LITERAL  elementOptions?
        |     range
        |    LEXER_CHAR_SET
        ;

block
        :    LPAREN  ( optionsSpec? ruleAction* COLON )? altList RPAREN
        ;

ruleref
        :    RULE_REF (LBRAC variableInitializer (COMMA variableInitializer)* RBRAC)?
             elementOptions?
        ;
range
        :  STRING_LITERAL RANGE STRING_LITERAL
        ;

terminal
        :    TOKEN_REF elementOptions?
        |    STRING_LITERAL elementOptions?
        ;

// Terminals may be adorned with certain options when
// reference in the grammar: TOK<,,,>
elementOptions  :    LT elementOption (COMMA elementOption)* GT
        ;

elementOption
        :    // This format indicates the default node option
             id
        |    // This format indicates option assignment
             id ASSIGN (id | STRING_LITERAL)
        ;

id   :    RULE_REF
     |    TOKEN_REF
     ;
```

Listing A.1: Context-Free Grammar of ANTLR's meta-language.

B

THE ANTLR TOOL

The purpose of this appendix is to give a short summary about the ANTLR v4 Tool[1], to allow the reader to quickly get to know the ANTLR tool, what it does and how it does it, and by this way to answer some questions that probably pop up in the reader mind while reading this document.

Quoting the authors, ANTLR is *" a powerful parser[2] generator that you can use to read, process, execute and translate a structured text or binary files"*Parr (2013). A parser is generated for the language specified by the input grammar, in other words, from the grammar a new program is created which is able to recognize valid sentences in the language described by the grammar. This input grammar must be written according with the syntax of ANTLR's meta-language.

With the generated parser, or syntax analyzer, an application can be built, responding with the desire behavior for each symbol that consumes or recognize. This application can be called *interpreter* if computes or *translator* if converts sentences from one language to another. This kind of programs have to recognize the symbols of the language and the order in which they appear. ANTLR splits the task in two different stages: the *Lexer*[3] an the *Parser*.

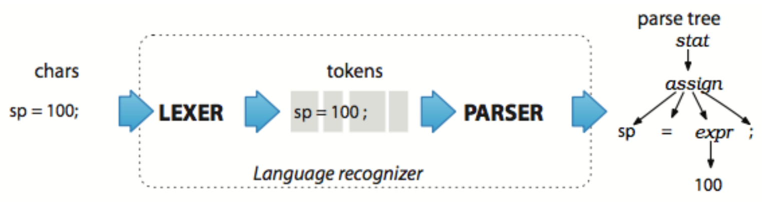


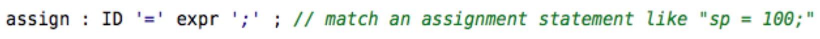
Figure 28.: Language Recognizer

The lexer, or lexical analyzer, tokenizes the input to feed the parser, that need the tokens to recognize the structure of the sentence. The way that the parser recognizes the structure of the input sentence and its component phrases are stored in a data structure built by the parser called *Parse Tree* or *Syntax Tree*.

1 Developed by Terrence Parr and Sam Harwell.
2 Program that breaks a sequence of terminal symbols of a given grammar into different parts (structure) according with the rules of the grammar that defines this language. Parsing means that the initial string is divided into smaller and specific components according to the structure of the language.
3 Separates the input character stream into vocabulary symbols of the language called *tokens*.

As shown in Figure 28, the input `sp = 100;` is recognized and the program knows that the input is an assignment statement, `sp` is the target and the value to store is `100`. The data structure stores in the internal nodes the rules of the grammar that were identified and store the tokens in the leaves of the parse tree are the terminals. Inspecting the parse tree in the illustrative figure, and without knowing the grammar, it can be said for sure that there is one rule in the grammar in which the non-terminal symbol `stat` derives in `assign`[4].

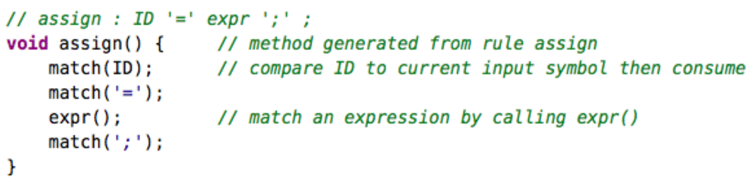
What really happens in the background of ANTLR, speaking in language processing terms, is that this tool generates *recursive-descent* parsers [5] from the grammar rules. Creating one method for each grammar rule, the parser enters in the first method (exactly the one that corresponds to the start symbol) and then it will make sure that all required tokens are present and in the order specified in the rule. If it gets a terminal simply has to match it with the token; if it gets a non-terminal the parser has to call the method with the name of that symbol. Considering again the example given in Figure 28, the corresponding `assign` rule is presented in Figure 29:



```
assign : ID '=' expr ';' ; // match an assignment statement like "sp = 100;"
```

Figure 29.: Simple `assign` rule in ANTLR.

For rule `assign`, ANTLR generate a method like the one listed in Figure 30:



```
// assign : ID '=' expr ';' ;
void assign() {      // method generated from rule assign
    match(ID);       // compare ID to current input symbol then consume
    match('=');
    expr();          // match an expression by calling expr()
    match(';');
}
```

Figure 30.: Method generated from the `assign` rule, to create the desired parser.

The idea is that when enters method `assign()`, the parser does not have to choose between more than one alternative. When that happens the parser have to predict which alternative will succeed to make a decision. This is possible by examining the next input token or *lookahead*. In some cases all the lookahead tokens have to be considered from the current position until the end of the file.

4 ANTLR provides some useful tools to traverse along the parse tree, such as walkers like *listeners* and *visitor*, enabling a more efficient application. This auxiliary tools are automatically generated by ANTLR and the same grammar can be reused in different application without another compilation. To find out more useful tools and tricks from ANTLR please consult Parr (2013).

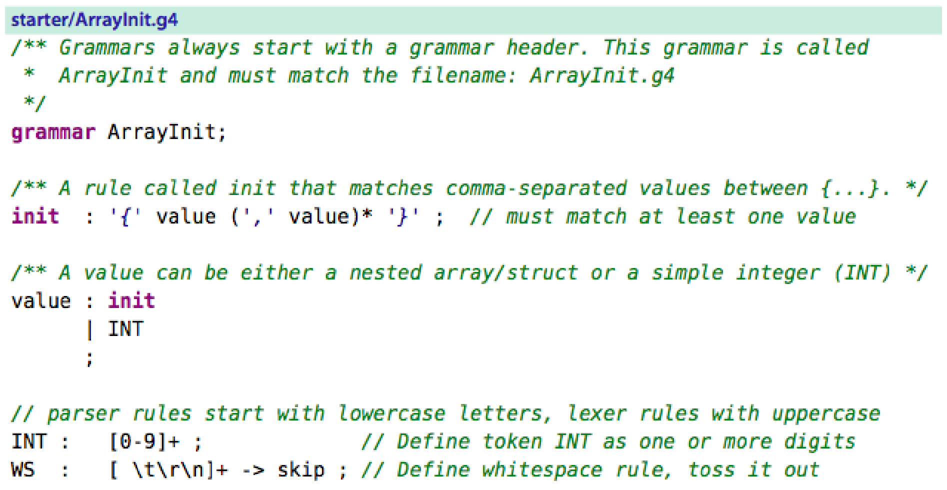
5 Recursive-Descent Parser is a kind of Top-Down parser, which means that the derivation tree is filled from the root (start symbol of the grammar) to the leaves, in other words, from top to bottom. The recursive term refers to the set of recursive methods generated, one per rule.

B.1 GENERATED CODE

As mentioned before, ANTLR tool automatically generate code for the user. This process is divided in two stages, working with different components of ANTLR in each one.

In the first stage, when it is said *"run ANTLR on a grammar"*, ANTLR tool itself is called to generate a parser and a lexer for the input grammar (*.g4* extension file), capable of recognizing sentences in the language described by the grammar. In the second stage, the generated files are compiled and executed with the ANTLR runtime API, which is a library of classes and methods required.

To see what really happens in the file system let´s explore the following example from Parr (2013). First a simple grammar is presented describing a simple array language, where the element are comma-separated values between {. . .}, as shown in Figure 31.



starter/ArrayInit.g4

```
/** Grammars always start with a grammar header. This grammar is called
 *  ArrayInit and must match the filename: ArrayInit.g4
 */
grammar ArrayInit;

/** A rule called init that matches comma-separated values between {...}. */
init  : '{' value (',' value)* '}' ;  // must match at least one value

/** A value can be either a nested array/struct or a simple integer (INT) */
value : init
      | INT
      ;

// parser rules start with lowercase letters, lexer rules with uppercase
INT :   [0-9]+ ;             // Define token INT as one or more digits
WS  :   [ \t\r\n]+ -> skip ; // Define whitespace rule, toss it out
```

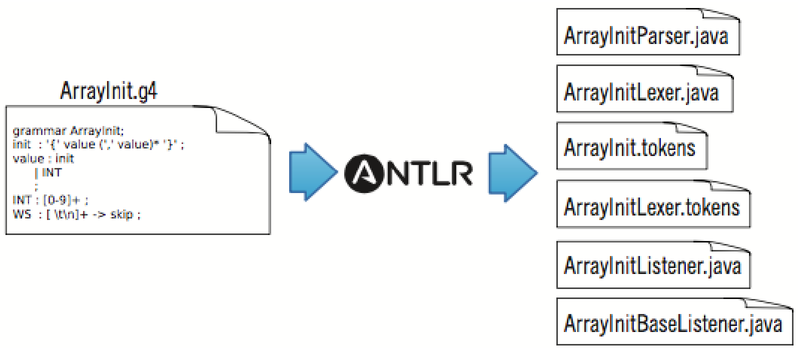
Figure 31.: Example of a simple Array language described in ANTLR format.

From the directory of this input grammar, the ANTLR tool (first stage) can be executed, by typing the following command[6]:

```
$ antlr4 ArrayInit.g4
```

Then some files are automatically generated, without any type of hand code, such as the ones displayed in Figure 32. Very briefly the generated files are: `ArrayInitParser.java` which contains the parser class definition, with one method for each rule, `ArrayInitLexer.java` which obviously contains the lexer class definition, `ArrayInit.tokens` that assigns a token type number for each token and store the values, `ArrayInitListener.java` is the *interface* that describes the callbacks for implementation, used by tree walkers and `ArrayInitBaseListener.java` is a set of empty methods that the user shall complete to create the *implementations*.

6 For more detailed information about the installation of the tool and how to make all the alias for the commands, please check the first chapter of Parr (2013).



```
grammar ArrayInit;
init  : '{' value (',' value)* '}' ;
value : init
      | INT
      ;
INT : [0-9]+ ;
WS  : [ \t\n]+ -> skip ;
```

Figure 32.: Generated files from *running* ANTLR on a *.g4* grammar.

Now to see the actual result of our generated parser, first the Java produced files have to be compiled, all of them, then the runtime library offers a useful test tool called *TestRig*, that can be invoked with a series of options like `-tokens` to print out the tokens that the lexer has created, `-tree` to print out the parse tree and the `-gui` option to visualize the tree in a dialog box. Following again the example and calling the runtime library with the name of the grammar file, the start symbol and with the `-gui` option, the result of the test is shown by Figure 33.

```
$ javac *.java
$ grun ArrayInit init -gui
{1,{2,3},4}
EOF[7]
```

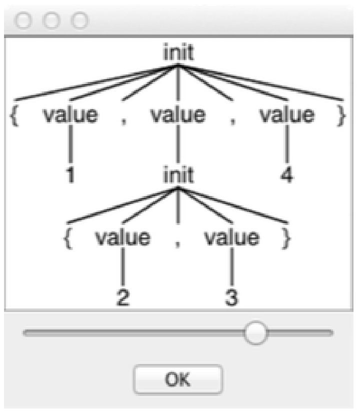
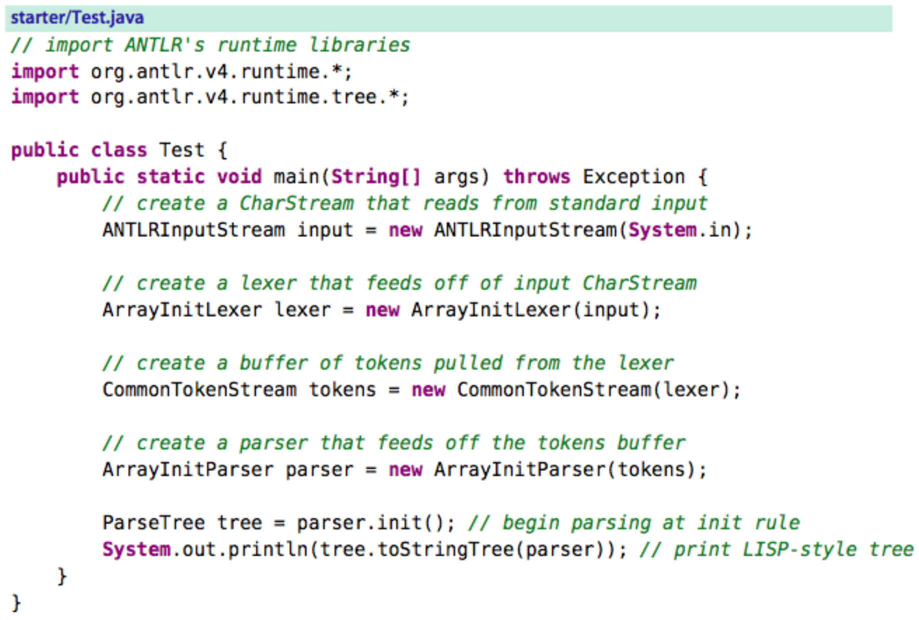


Figure 33.: Dialog box exited by ANTLR with `-gui` option with the filled parse tree.

Of course this is a simple example of using ANTLR to generate the parser and test the grammar. When using this kind of tool in a language application, like the one presented in this document, it is useful to integrate this code in a main program, written in Java. This was the technique followed to

7 Ctrl+D on Unix and Ctrl+Z on Windows.

execute the parser generated by the grammar described in the previous chapters. Just to clarify, let's take a look at Figure 34, an example provided by Parr (2013), and followed in the development of *GQE*. The only difference is that in this example after the parsing, the parse tree in printed out, such as the `-tree` command line option.



starter/Test.java

```java
// import ANTLR's runtime libraries
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input
        ANTLRInputStream input = new ANTLRInputStream(System.in);

        // create a lexer that feeds off of input CharStream
        ArrayInitLexer lexer = new ArrayInitLexer(input);

        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        // create a parser that feeds off the tokens buffer
        ArrayInitParser parser = new ArrayInitParser(tokens);

        ParseTree tree = parser.init(); // begin parsing at init rule
        System.out.println(tree.toStringTree(parser)); // print LISP-style tree
    }
}
```

Figure 34.: Integration of ANTLR tool in a Java main program.

B.2 ANTLR GRAMMAR FORMAT

In this section will be provided a short description about ANTLR meta-language. Not all aspects of the language will be covered, but the information given should be enough to understand the grammar language and how to create one.[8]

First the lexicon of ANTLR meta-language will be presented, followed by the structure of an ANTLR grammar. Then the syntax options of parser rules, lexer rules and finally actions and attributes needed for Attribute Grammars.

B.2.1 *Lexicon*

The lexicon of ANTLR is very similar to most of programming languages, such as C, aside the necessary grammatical extensions. In a very short explanation the lexicon of ANTLR allows:

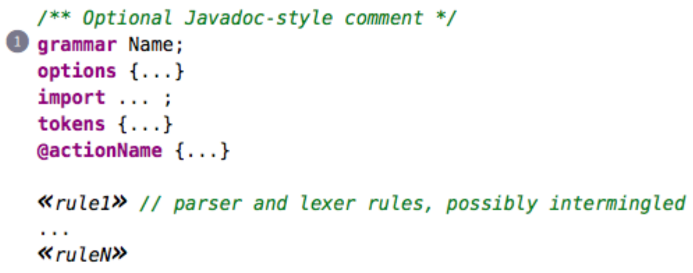
- **Comments** - Single line (// . . .), multiline (/*. . . */) and Javadoc-style comments (/**. . . */);

8 Once more, for more detailed information about this subject please take a look at chapter 15 from Parr (2013).

- **Identifiers** - Token names and lexer rules start with a capital letter, on the other hand parser rules must start with lowercase letters. The the following characters can be uppercase or lowercase letters, digits and underscores;
- **Literals** - All literal strings that have one or more characters in length are enclosed in single quotes. They can contain Unicode escaped sequences and also escape sequences like newline ('\n'), carriage return ('\r'), tab ('\t'), backspace ('\b') and form feed ('\f');
- **Actions** - Code block written in the target language, specified in the `language` option. The code must be surrounded by curly braces and can appears in a lot of places on the grammar;
- **Keywords** - The list of reserved keywords are: `import`, `fragment`, `lexer`, `parser`, `grammar`, `returns`, `catch`, `finally`, `mode`, `options` and `tokens`.

B.2.2 *Structure*

The syntax of a grammar in ANTLR is very simple, first there is a list of declarations(all of them are optional except for grammar name) and then a set of rules. To write a grammar there is only two things that are required: the grammar name declaration (represented by the header **1** in Figure 35) and a non-empty set of rules.



```
/** Optional Javadoc-style comment */
grammar Name;
options {...}
import ... ;
tokens {...}
@actionName {...}

«rule1» // parser and lexer rules, possibly intermingled
...
«ruleN»
```

Figure 35.: General form of a grammar written in ANTLR.

The filename that describes grammar *X* must be called *X*.g4. The header declaration can be used also to specify the use of separated grammars (parser and lexer), by simply affix the keyword (`parser` or `lexer`) in the start of the declaration. Another useful feature is to import another grammars with the `import` declaration[9], lexer grammars can include lexer grammars, parser grammars can include parser grammars and combined grammars can include lexer or parser grammars.

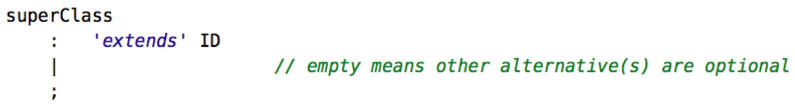
Tokens section is used to define some tokens not declared in the lexer to add to the overall set. As for the actions section, they are used to execute actions at a grammar level, just for Java target language:

9 ANTLR leads with importing a grammar kinda like object-oriented languages leads with super classes. A grammar can inherit all the rules, tokens specifications and named actions from another one. Rules on the *main* grammar override the rules from imported grammars.

`@header`(for importing java packages or classes) and `@members` (to declare global variables or objects that can be accessed from any part of the grammar).

B.2.3 *Parser Rules*

Parser rules are very easy to read, write and understand in ANTLR. Rules are written in the same way that the production are read, just as stated in the section 2.3 of the chapter 2, with some simple differences. When exists more than one production for the same non-terminal, this productions must be all written in one parser rule where alternatives are used from the second until the last production of that non-terminal.



```
superClass
    :   'extends' ID
    |                   // empty means other alternative(s) are optional
    ;
```

Figure 36.: Parser rule example in ANTLR.

Examining the figure 36 example, there are two productions from the same non-terminal and the rule must be read like *"The symbol `superClass` derives in (`':'`) the reserved word `'extends'` (string literal) followed by the terminal `ID`,* ***or*** *(`'|'`) in an empty alternative"*. This match the formal declaration:

$p_x : SuperClass \rightarrow "extends"$ id

$p_y : SuperClass \rightarrow \&$

All rules in ANTLR must end with `';'` and it is possible to do a lot of tricks such as labeling the alternatives, labeling the elements of a rule and inserting subrules.[10] In the right hand side of a rule is possible to appear tokens (starts with uppercase letter), string literals, non-terminals for other rules(starts with lowercase letter) and actions[11]. When working with attribute grammars it is possible to also appear something like `r[≪args≫]`, which match a rule passing in a comma-separated list of expressions representing the inherited attributes for rule `r`.

B.2.4 *Actions and Attributes*

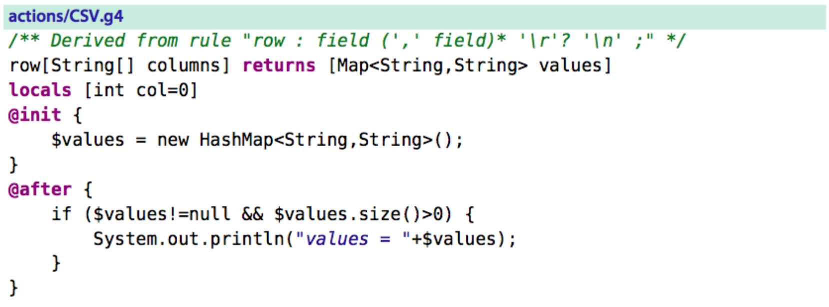
Just like methods in programming languages, rule can have arguments, return values and local variables. The arguments, values and variables are usually called *attributes* and follow the general syntax:

`rulename[≪args≫]` **`returns`** `[≪retvals≫]` **`locals`** `[≪args≫]:  ...;`

10 Alternative blocks on the right hand side of the rules, in which the alternatives are surrounded by parentheses and followed or not by a Extended-BNF operator (?, * or +), for example (x|y|z)?, or (x|y|z)*.

11 Block of code written in the target language and surrounded by `{...}`. Instructions are executed exactly on that position before the recognition of the following symbol.

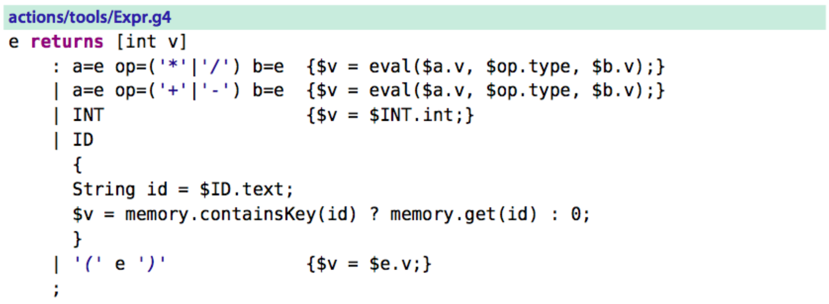
Figure 37 demonstrates precisely that case in which the rule `row` receive one attribute, an array of String named `columns` as argument and returns a Map collection named `values`, where the keys are Strings as well as their respective values. Also, a local variable `col` is declared and initialized. In a rule-level is also possible to specify actions.



```
actions/CSV.g4
/** Derived from rule "row : field (',' field)* '\r'? '\n' ;" */
row[String[] columns] returns [Map<String,String> values]
locals [int col=0]
@init {
    $values = new HashMap<String,String>();
}
@after {
    if ($values!=null && $values.size()>0) {
        System.out.println("values = "+$values);
    }
}
```

Figure 37.: Declaration of attributes and local variables in a rule.

As is easily noticed in the previous figure, another important aspect is the syntax *$attribute* to access or modify the value of an attribute, that is declared **inside** the rule itself. To access an attribute that is associated with another parser rule, there is only one way to do it: *$r.attribute* where *r* is the rule name or a label assigned to a rule reference. All three possibilities are exposed in the following figure.



```
actions/tools/Expr.g4
e returns [int v]
    : a=e op=('*'|'/') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | a=e op=('+'|'-') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | INT                   {$v = $INT.int;}
    | ID
      {
      String id = $ID.text;
      $v = memory.containsKey(id) ? memory.get(id) : 0;
      }
    | '(' e ')'             {$v = $e.v;}
    ;
```

Figure 38.: Accessing attributes and local variables in a rule.

B.2.5 *Lexer Rules*

Lexer rules specify tokens definitions and have a similar syntax of parser rules, except for the no existence of arguments, return values or local variables. ANTLR provide a lot of ”gadgets” for lexer rules such as lexical modes, recursive rules, lexer rule actions and lexer commands.

On the right hand side of lexer rules can appear:

literal - Match that sequence of characters;

[char set] - Match one of the characters in the character set. Interpret *x-y* as a set of characters between range *x* and *y*, inclusively;

'x' ... *'y'* - Match any character from character *x* to *y*;

T - Summon lexer rule *T*;

. - The dot match any single character (wildcard);

$\{\ll$*action*$\gg$´$\}$ - Actions once again written in the syntax of the target language;

x - Match any single character not in the set described by *x*.

C

NATURAL LANGUAGES TOOLS

In this appendix are presented three auxiliary programs used on the development of GQE application. All these programs are inserted in the Natural Language processing tools family and were used to essentially help the assessment of Lexicographic metrics defined in the chapter 2 and implemented by GQE.

Two of the program are perl modules, because the component in which they are integrated it is written in the Perl language, and one is a stand alone tool.

C.1 PERL MODULE LINGUA::IDSPLITTER

Lingua::IdSplitter[Carvalho et al. (2015)] is a perl module designed to split source code identifiers into words. Was developed by Nuno Carvalho and implemented in the SplitterExpander component, with some modifications made by the author. The purpose of this section is to introduce this module, justifying why were he used, why he have to be modified and the benefits of those modifications.

In simple words and quoting, Lingua::IdSplitter is "*a dictionary based algorithm for splitting and expanding strings that compose multi-term identifiers*"[Carvalho et al. (2015)]. This original expansion is based on custom dictionaries for programming language terms, abbreviations, acronyms and on a natural language dictionary[1]. The words splitting is made by searching for two patterns: '_' separator and CamelCase separator. This techniques are used for programmers to describe muti-term identifiers. If non of the two splitting techniques is found and the word as no immediate expansion, then he tries to separate terms letter by grouping letters.

Clearly, and taking into account the definition of lexicographic metrics made in Section 2.3, this module was a very useful choice for the metrics implementation in GQE. His main feature coincide with the desire to verify, in the grammars, if all identifiers ***derives*** from a concept name or not.

However, some modifications were made to improve, the already satisfying, results of this module. The intention was to make it more suitable with a grammar application and to increase, particularly, the expansion succession rate feature.

The modifications are:

1 This dictionary is achieve by creating a *Text::Aspell* speller from one natural language, in this case, English. For more details please consult http://search.cpan.org/ hank/Text-Aspell/Aspell.pm

- two more custom dictionaries were added, one for easy expansions of terms related with grammar domain concepts and other with easy expansions related with inline, block and meta-information comments;
- only two multi-terms separation techniques are valid: the '_' separator and CamelCase separator. Because of the rules stated in Definition 12, the technique were Lingua::IdSplitter search for multi-terms by grouping different sequences of letters with the help of a rating system, have to be eliminated. If the identifier have multi-terms and not separate them with some technique, clearly make it invalid to be understanded;
- some values for rating were changed too, because to assess correctly some metrics, some information about how expansions were achieved must be presented. The idea was using the own rating system to perform that information.

With this medications, most of the lexicographic metrics were evaluated successfully, bringing to the application vital information about the syntax and the semantic of the identifiers presented in the grammar. The only downside of this choice is the modifications of the original Lingua::IdSplitter module.

C.2 WORDNET

Wordnet is a tool for computer linguistics and natural language processing supported by a large lexical database of the English Language. Words are divided in lexical categories such as nouns, verbs, adverbs and adjectives, and grouped together into sets of cognitive synonyms called **synsets**.

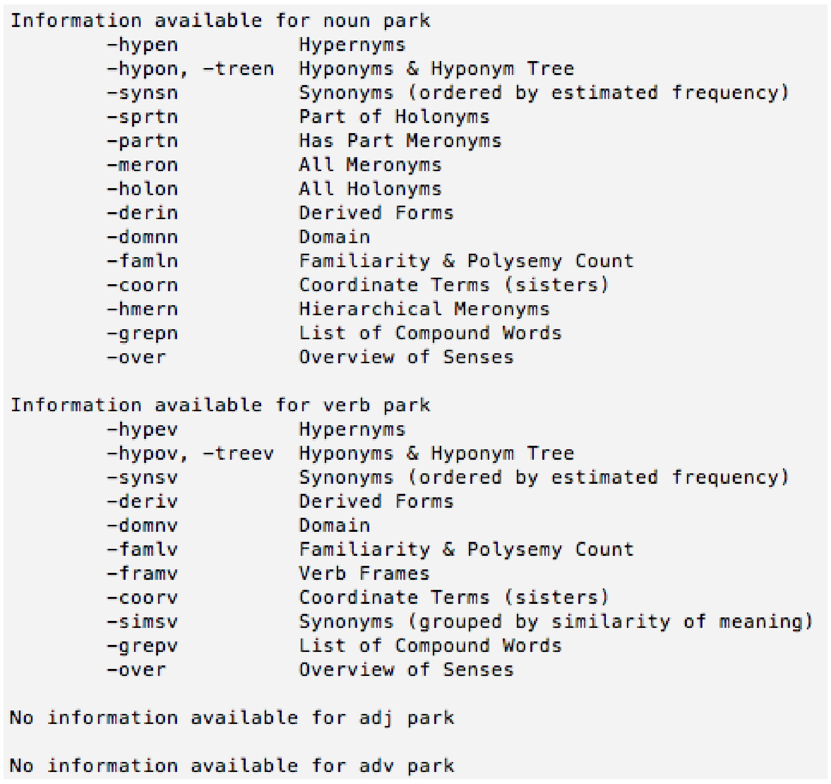
All synsets are connected with each others by a few conceptual **relations**. They provide a brief definition (*gloss*) and, in most cases, one or more short sentences illustrating the use of the synset members.

Sometimes, words can have different senses causing the assigning to different synsets. The main relations among words is synonyms because they denote the same concept, then according to the lexical category and the meaning of the word more relations are presented. An example of that kind of polysemous words is *'park'*, like is shown in Figure 39 where the word is simply passed to wordnet.

As is seen, wordnet found the word *park* in two different lexical categories: nouns and verbs. Now, to check the senses of the word in each category, the *overview* options is passed to wordnet, with the following result, exposed in Figure 40.

C.2.1 *Relations*

The most common relation among synsets is the super-subordinate relations also known as hypernymy, hyponymy or IS KIND OF relation, linking more general synsets to more specific ones. All nouns



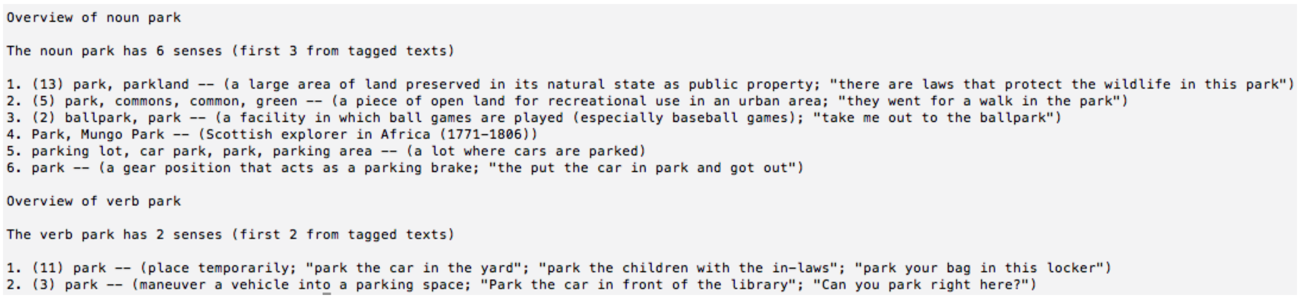
```
Information available for noun park
        -hypen          Hypernyms
        -hypon, -treen  Hyponyms & Hyponym Tree
        -synsn          Synonyms (ordered by estimated frequency)
        -sprtn          Part of Holonyms
        -partn          Has Part Meronyms
        -meron          All Meronyms
        -holon          All Holonyms
        -derin          Derived Forms
        -domnn          Domain
        -famln          Familiarity & Polysemy Count
        -coorn          Coordinate Terms (sisters)
        -hmern          Hierarchical Meronyms
        -grepn          List of Compound Words
        -over           Overview of Senses

Information available for verb park
        -hypev          Hypernyms
        -hypov, -treev  Hyponyms & Hyponym Tree
        -synsv          Synonyms (ordered by estimated frequency)
        -deriv          Derived Forms
        -domnv          Domain
        -famlv          Familiarity & Polysemy Count
        -framv          Verb Frames
        -coorv          Coordinate Terms (sisters)
        -simsv          Synonyms (grouped by similarity of meaning)
        -grepv          List of Compound Words
        -over           Overview of Senses

No information available for adj park

No information available for adv park
```

Figure 39.: Wordnet usage example, with the word *park*.



```
Overview of noun park

The noun park has 6 senses (first 3 from tagged texts)

1. (13) park, parkland -- (a large area of land preserved in its natural state as public property; "there are laws that protect the wildlife in this park")
2. (5) park, commons, common, green -- (a piece of open land for recreational use in an urban area; "they went for a walk in the park")
3. (2) ballpark, park -- (a facility in which ball games are played (especially baseball games); "take me out to the ballpark")
4. Park, Mungo Park -- (Scottish explorer in Africa (1771-1806))
5. parking lot, car park, park, parking area -- (a lot where cars are parked)
6. park -- (a gear position that acts as a parking brake; "the put the car in park and got out")

Overview of verb park

The verb park has 2 senses (first 2 from tagged texts)

1. (11) park -- (place temporarily; "park the car in the yard"; "park the children with the in-laws"; "park your bag in this locker")
2. (3) park -- (maneuver a vehicle into a parking space; "Park the car in front of the library"; "Can you park right here?")
```

Figure 40.: Wordnet usage example, with the word *park* and the *overview* option.

hierarchies starts with the root node (entity) and both relations are transitive. Nouns are distinguished between Types and Instances[2].

Another important relation is meronyms or the part-whole relation, such as between *chair* and *backrest*, *seat* and *leg*. Parts are inherited from their superordinates, but not from their subordinates: *chairs* and kinds of chairs have *legs*, but not all kinds of *furniture* have *legs*.

2 Wordnet even distinguish specific persons, countries and geographic entities, such as Barack Obama is an instance of a president.

Just like nouns, verbs are also grouped in hierarchies, where the verbs at the bottom express a specific action (move–jog–run), always depending on the semantic field. Even verbs with an indirect relation are linked together such as (buy–pay) or (succeed–try).

Adjectives are linked in terms of antonyms or semantic similarity, such as (dry–wet) for the first case and (dry–arid,parched) for the second one. Finally the adverbs are in short supply, as most of English adverbs are straightforwardly derived from adjectives via morphological affixation (surprisingly, strangely, etc.).

Speaking know in practical terms of what can be passed to the wordnet tool and as stated before, synsets are interlinked between them by means of semantic relations and these relations are not all shared by all lexical categories.

C.2.2 *Options*

Using the command line interface **wn** for Wordnet lexical database, some useful options are provided, which make the wordnet a precious tool for a lot of different applications. The syntax of the command line call is:

wn [searchstr] [-h] [-g] [-a] [-l] [-o] [-s] [-n#] [search_option . . .]

Forgetting the normal flags, let us focus on what can be really extracted, changing the *search_options*, from wordnet database. Please note that the relation is available only for some lexical categories, as shown in the information between parentheses where, **n** stands for nouns, **v** for verbs, **a** for adjectives and **r** for adverbs.

- **syns** (n | v | a | r) — Display synonyms and immediate hypernyms of synsets containing searchstr .
- **simsv** — Display verb synonyms and immediate hypernyms of synsets containing searchstr . Synsets are grouped by similarity of meaning.
- **ants** (n | v | a | r) — Display synsets containing antonyms of searchstr .
- **faml** (n | v | a | r) — Display familiarity and polysemy information for searchstr .
- **hype** (n | v) — Recursively display hypernym (superordinate) tree for searchstr (searchstr IS A KIND OF _____ relation).
- **hypo** (n | v) — Display immediate hyponyms (subordinates) for searchstr (_____ IS A KIND OF searchstr relation).
- **tree** (n | v) — Display hyponym (subordinate) tree for searchstr . This is a recursive search that finds the hyponyms of each hyponym.
- **coor** (n | v) — Display the coordinates (sisters) of searchstr . This search prints the immediate hypernym for each synset that contains searchstr and the hypernym's immediate hyponyms.

- **deri** (n | v) — Display derivational morphology links between noun and verb forms.
- **domn** (n | v | a | r) — Display domain that searchstr has been classified in.
- **domt** (n | v | a | r) — Display all terms classified as members of the searchstr 's domain.
- **subsn** — Display substance meronyms of searchstr (HAS SUBSTANCE relation).
- **partn** — Display part meronyms of searchstr (HAS PART relation).
- **membn** — Display member meronyms of searchstr (HAS MEMBER relation).
- **meron** — Display all meronyms of searchstr (HAS PART, HAS MEMBER, HAS SUBSTANCE relations).
- **hmern** — Display meronyms for searchstr tree. This is a recursive search that prints all the meronyms of searchstr and all of its hypernyms.
- **sprtn** — Display part of holonyms of searchstr (PART OF relation).
- **smemn** — Display member of holonyms of searchstr (MEMBER OF relation).
- **ssubn** — Display substance of holonyms of searchstr (SUBSTANCE OF relation).
- **holon** — Display all holonyms of searchstr (PART OF, MEMBER OF, SUBSTANCE OF relations).
- **hholn** — Display holonyms for searchstr tree. This is a recursive search that prints all the holonyms of searchstr and all of each holonym's holonyms.
- **entav** — Display entailment relations of searchstr .
- **framv** — Display applicable verb sentence frames for searchstr .
- **causv** — Display cause to relations of searchstr .
- **pert** (a | r) — Display pertainyms of searchstr .
- **attr** (n | a) — Display adjective values for noun attribute, or noun attributes of adjective values.
- **grep** (n | v | a | r) — List compound words containing searchstr as a substring.

C.3 PERL MODULE WORDNET::QUERYDATA

This perl module appears in this application just as the interface or API, for the previous described natural language tool *Wordnet*. In this section will be explained how this module is implemented in the SplitterExpander program and what kind of information is retrieved from the *Wordnet* database semantic lexicon.

Since, the SplitterExpander component, introduced in Chapter 4, Section 4.5, is written in perl programming language, makes sense to use this perl module as a bridge between the two applications: SplitterExpander and *Wordnet*. This option makes the component more clear and efficient, because this module is easy to learn, with few functions and options, and there was no need to think in some always complex communication mechanism to deal with the stand alone *Wordnet* tool.

Essentially, to understand minimally how this module was implemented in the SplliterExpander perl program, there are a few things to know:

- this module provide a series of methods to interact with *Wordnet*, but only two of them are used to query the database: *querySense* and *queryWord*. The first is for extracting semantic information (sense to sense) relations and the second is for lexical relations. Only the first form of querying **querySense** is used in the component, because we are looking for semantic relations, words related with the grammar domain, and not syntactic relations.

- for this query process, we have to specify one word and one relation. There are three ways to specify word: *i*) word *ii*) word#pos (pos represents the lexical categories introduced in Section C.2) *iii*) word#pos#sense . The first two types require no relation to query the database and will return query forms available. The third type requires a relation and that is when the related words are returned.

  As for the relations they were exposed in the previous section, but only some of them were used in the implementation. Some relations can only be sent to queryWord and others to querySense.

To implement this module in the SplitterExpander perl program, a method must be used, just like is shown in Listing C.1. First a querySense is made with grammar domain. The result will give all the available **pos**, related to the specified domain. Next, for each one, a new querySense is made, resulting again in another set of available queries, but this time for **senses**. The same process is made again, this time with type *iii*) queries, giving back the desire list of words.

```
my $wn = WordNet::QueryData->new( dir => "/usr/local/Cellar/wordnet/3.1/dict/",
    noload => 1);
my @relations = ('glos','syns','hypes','hypos','mero','holo','domn','domt');
my @pos = $wn->querySense($grammar_domain);
my @senses;

foreach(@pos){
        push @senses , $wn->querySense($_);
}

my @words;
foreach my $s(@senses){
        foreach my $r(@relations){
                push @words , $wn->querySense($s,$r);
        }
}
```

Listing C.1: Snippet from SplitterExpander component related with the Wordnet::QueryData implementation.

For more details about this module and others features that he present, please consult the CPAN page http://search.cpan.org/ jrennie/WordNet-QueryData-1.49/QueryData.pm.