Determining Programming Languages Complexity and Its Impact on Processing

Gonçalo Rodrigues Pinto
Department of Informatics, University of Minho, Braga, Portugal

Pedro Rangel Henriques
Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Daniela da Cruz
Checkmarx, Braga, Portugal

João Cruz
Checkmarx, Braga, Portugal

Abstract

Tools for Programming Languages processing, like Static Analysers (for instance, a Static Application Security Testing (SAST) tool), must be adapted to cope with a different input when the source programming language changes. Complexity of the programming language is one of the key factors that deeply impact the time of giving support to it.

This paper aims at proposing an approach for assessing language complexity, measuring, at a first stage, the complexity of its underlying context-free grammar (CFG). From the analysis of concrete case studies, factors have been identified that make the support process more time-consuming, in particular in the stages of language recognition and in the transformation to an abstract syntax tree (AST). In this sense, at a second stage, a set of language characteristics is analysed in order to take into account the referred factors that also impact on the language processing.

The principal goal of the project here reported is to help development teams to improve the estimation of time and effort needed to cope with a new programming language. In the paper a tool is proposed, and its prototype is presented, that allows the evaluation of the complexity of a language based on a set of metrics to classify the complexity of its grammar, along with a set of properties. The tool compares the new language complexity so far determined with previously supported languages, to predict the effort to process the new language.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Complexity, Grammar, Language-based-Tool, Programming Language, Static code analysis

Digital Object Identifier 10.4230/OASIcs.SLATE.2022.16

Supplementary Material *Software (Web Application)*: `https://lce.di.uminho.pt/`
archived at `swh:1:dir:ec41f17cb7b247b4615a92cf8fc37b82b3fc972c`

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

Acknowledgements We want to thank the reviewers for the input and suggestions on the paper.

1 Introduction

A SAST tool analyses source code written in a programming language and finds its security vulnerabilities. While this solution satisfies the need (detecting software vulnerabilities), there are other factors that need special attention in this type of tool, one of which is the maintenance required.

Several new practices have emerged in recent years that can improve software maintenance. The major consideration is how to balance the enormous complexity of software with its cost, effort, and time required for maintenance. For that, it must be adapted to handle different inputs when the source programming language varies.

To do this, one of the first steps towards supporting a new programming language in this tool is to create a new parser to analyse the relevant language.

The complexity of the programming language is one of the key factors that affects the time to provide support for it. This limitation raises the need to evaluate whether the complexity of a programming language is related to the complexity of its context-free grammar.

Thus, given the difficulties associated with the SAST engine in analysing and supporting a new programming language, it is motivating to create a tool that selects and implements a set of metrics and analyses a set of properties that allow us to assess the complexity of a language.

The primary purpose of the study described here is to assist language support teams in better estimating the time and effort required to support a new programming language. Along the paper, we propose and present a tool for evaluating the difficulty of supporting a language based on a collection of metrics to classify the complexity of its grammar, as well as a set of properties. To forecast the work required to process the new language, the program compares the new language difficulty so far identified with previously supported languages.

This Section 1 discussed the significance of maintenance, what a SAST tool is and its limits, how complexity is to be measured, and why the provided tool was developed. In Section 2, it is intended to focus on the main points to characterize the concepts of software, language, and grammar in determining the complexity of programming languages and their impact on processing. After the concepts have been introduced, Section 3 follows, in which the DSL created for this purpose is presented in order to represent the extra-grammatical characteristics that have to be described by those who know the language. Introduced and described the language intended for this particular problem domain, it is fundamental to talk about the proposal to be developed, showing its architecture and the results already obtained to produce a quantitative and qualitative report of the language, this information is described in the Section 4. Finally, Section 5 is the summary of the document, some conclusions and results achieved, and a description of future work.

2 Software, Grammar, Language Complexity and the impact on processing

Section 2 begins by introducing the concept of software complexity and its impact on the timing of support. After that, one of the tools that allows to evaluate the complexity of a language and grammars, is presented, explaining its relation with languages and how grammatical complexity is defined. Afterwards, the way to measure this grammatical complexity, by metrics, is presented. Finally, the subject of this project, complexity of programming languages, is introduced.

2.1 Software Complexity

Knowledge about the properties of entities is obtained through measurement. In order to relate and compare properties between entities, rules are used. Nevertheless, measurement is not something clear or easy to define, because it is always open to subjective interpretation. Every time we effectively measure something that was not measurable at first glance, we expand the power of software engineering, as is done in other disciplines in this area.

There is no theory that shows whether a set of metrics is valid. We only know that there is a structure based on objectives for software measurement, which can improve software engineering practices.

This structure is based on three principles: categorizing the entities to be investigated, determining relevant measuring targets, and determining the maturity level attained.

In recent years, software complexity has been the subject of much interest in order to define measures for measuring it. Complexity is the characteristic associated with a system or model whose state is composed of many parts and is difficult to understand or find an answer for. Understanding and measuring the software complexity is not something simple and obvious.

However, measuring the complexity of the problem associated with this software is useful, as it may prevent the effort or resources needed for the project. By comparing the problems and considering the solutions found for the problems already solved, it is possible to predict the properties of the new solution to the latest problem, such as cost or time.

Size along with structure are the main internal properties in measuring software complexity, according to Fenton and Pfleeger in 1998 [6].

- **Size Complexity** – the traditional attribute to measure in software, because it is advantageous, accessible to measure without having to run the system, and because software development is a physical entity.
- **Structure Complexity** – determines the level of project productivity, as it has been proven that a larger module does not always take longer to specify, design, code, and test than a small one. The structure of the product affects its maintenance and development effort.

Therefore, complexity can be assessed by quantifying a subset of software metrics that are based on static analysis. In this way, we can better understand the language in some aspects, such as the size and structure.

2.2 Grammar Complexity

Since any grammar characterizes a language and gave a premise for determining elements of that language, a grammar might be considered as both a program and a specification. Grammars formally specify languages, so the complexity of languages depends on the complexity of grammars, even if the complexity of grammars does not fully imply the complexity of language analysis.

In this context, the use of grammars is proposed to define the languages and support their recognition, which leads to a strong relationship between grammar and the language that is defined by that grammar [7]. Therefore, grammar will be one tool to assess the complexity of a language.

Considering what has been previously presented to show the relationship between grammars and languages, supporting a new programming language in a static analysis tool is faster and requires less effort, the less complex the grammar is.

The complexity of a grammar as a characterizer and producer of a language that directs the recognition of sentences in that language concerns how the symbols depend on each other, i.e., the number of symbols on the right-hand side of a production for a given symbol on the left-hand side, or how many symbols that symbol intervenes in.

Considering this, the need to evaluate the complexity of a grammar arises, since it will allow us to evaluate the complexity of the language defined by it. Thus, the use of grammatical metrics is relevant to the study in question.

2.2.1 Measuring Grammar Complexity

The metrics for evaluating the complexity of a well-formed context-free grammar are presented, dividing them into the previously mentioned criteria:

- Size metrics that measure the number of symbols (terminals or non-terminals) and productions used to write the grammar. As the grammar is the basis to recognize the sentences of the language defined by itself, it is reasonable to state that the size of the grammar has a direct impact on the time and effort necessary to support that language
  Size Metrics

Table 1 Metrics for evaluating the Size of Context-Free Grammars.

Metric	Definition
#P	Number of productions
#N	Number of non-terminals
#T	Number of terminals
#UP	Number of unit productions
RHS-Max	Maximum number of symbols on an RHS
RHS	Average number of symbols in the RHS
ALT	For the same left sides, average size of alternative productions
MCC	McCabe cyclomatic complexity

- Structure metrics that measure the dependency among the symbols of a grammar induced by its productions. Once again, we can state that the more intricate are the interrelations among the symbols, the harder it is to support the grammar and to recognize the sentences of the generated language. To compute those metrics, a grammar is represented as a graph.
  Structure Metrics

Table 2 Metrics for evaluating the Structure of Context-Free Grammars.

Metric	Definition
#R	Number of recursive symbols
FanIn	Average number of branches of the input nodes (non-terminals) of the DGS
FanOut	Average number of branches of the output nodes of the DGS
TIMP	Tree Impurity
CLEV	Normalized Counts of Levels
NSLEV	Number of Non-Singleton Levels
DEP	Size of The Largest Level

2.3 Language Complexity

Software security is turning into an inexorably significant differentiator for IT organizations. Therefore, methods for forestalling software vulnerabilities during software development are turning out to have increasing significance. The longer it takes to find the vulnerabilities, the more costly it will be to fix, and making an already difficult situation even worse.

In order to identify existing vulnerabilities, Static Application Security Testing, abbreviated as SAST and often alluded to as “White-Box Testing”, is used. The tool performs a security test that examines the source code of applications.

The idea behind this type of analysis is to identify in the code the use of language constructions that are vulnerable and can facilitate external attacks on the SW system. Static analysis examines the text of a program statically, without running it. In Figure 1, a high-level description of the scanning pipeline that allows for the aforementioned static analysis of the source code will be presented.

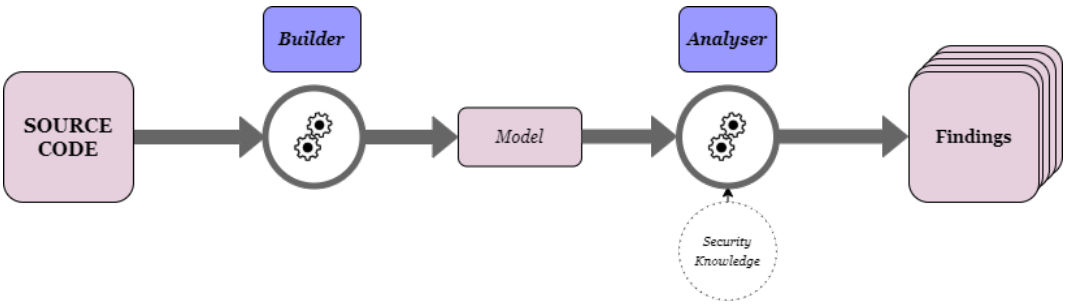

Figure 1 An abstract, high-level model of a SAST tool.

In a comprehensive way, a SAST tool basically comprises two processes.

1. Model **Builder** by ingesting the source code and transforming it into a normalized, understandable model for the analyser to decipher. A typical model is the development of an AST, a simplified representation of the structure of the source code, where each node in the tree is associated with a constructor of the code. In this way, the syntax is abstract since it does not represent every detail, as it appears in real syntax.
2. **Analyser based on existing security knowledge**, using a series of rules to figure out what the tool should evaluate within the source code. It is critical to customize and calibrate these rules to suit a specific application, as this allows for more reliable and worthwhile results. The feasibility of these methods is the main inner component of a SAST tool, from a customer's point of view.

The analyser based on current security knowledge process is irrelevant to the goal project since the challenge of supporting a new programming language resides in the model's builder process, because the language in which the source code is produced influences the static analysis process.

Taking this into consideration, the complexity of a language is defined by the complexity of the grammatical structure and the semantic properties, these two factors thus highlight the time and effort needed for support.

In this sense, the complexity of supporting a new language in this case study is strongly related to the AST generated [8]. For this reason, for the support to be more productive, at the level of less time and effort, the programming language in which the code under analysis is written, should generate an AST with the following characteristics:

- **Dense**: No pointless nodes;
- **Convenient**: Patterns in the tree are straightforward and quick to discern;
- **Significant**: Emphasize operators, operands, and the connection between them.

The initial two focuses infer that it ought to be simple and quick to identify patterns in the tree. In order to gather as much useful information as possible about each node, it is usually necessary to make multiple passes over the tree.

The last point infers that the tree structure ought to be insensitive to changes in the grammar.

Besides this strong relationship, there is another aspect that the complexity of the programming language influences, since when the tree is generated there are properties of the nodes that have not been assessed so far and are necessary in order to detect as many vulnerabilities as possible.

In general, it is intended that each node of the AST contains the following properties:

- **Name**: Symbols are identifiers like x and y, yet they can be operators as well, as for example, the symbol known as the addition operator (+);
- **Category**: The thing like the symbol is. Is it a class, method, variable, name, etc. To approve a method called x + y, for instance, we want to realize that + is a method to add 2 or more values, not a variable or class.
- **Type**: When the symbol is a variable, there is an interest in knowing what type it concerns, in order to be able to subsequently approve certain operations. Normally, the software engineer needs to explicitly distinguish between each type of symbol (in some languages, the compiler assumes this).
- **Scope**: The scope of a symbol restricting is the part of a program where the symbol is valid, that is, the place where the symbol can be utilized to refer to the element.

The cost of supporting a programming language in a SAST tool is a function of many of its properties. The following is one of the main characteristics[1] present in a real programming language that influences the complexity of support in this type of application.

Declaration in a programming language is a statement that determines the properties of a symbol. A declaration introduces a program Entity identified by a unique name (an identifier), its category (function, variable, constant, etc.), its type (in case it is a variable or constant) or if it is a function what it accepts as input and output, it can also declare things like the size of a type.

▶ **Example 1.** *Consider for example the same code written in different programming languages, Python, Java, JavaScript, respectively, to declare the same variables and the same function.*

```python
def main():
    x = 13
    y = "Python!"
```

```java
public class Main {
    public static void main(String[] args) {
        int x = 13;
        String y = "Java!";
    }
}
```

```javascript
function main() {
    let x = 13;
    var y = "JavaScript!"
}
```

A declaration conveys the "meaning" of a symbol, which highlights that this property is related to semantics and not syntax. However, there is interest in analysing this because of the extra effort needed in the Builder process.

[1] In order to make the document as compact as possible no further features will be described, but the reader is warned that there are other relevant properties.

The fact that a declaration is used to communicate the presence of an entity to the compiler means that, in dynamically typed languages such as Python, it is unnecessary to specify the variables; the runtime interpreter does the verification work. Considering the present case study, as shown, a static analysis tool does not execute any code. For that reason, in order to detect as many vulnerabilities as possible, it needs to do the work of the compiler.

In comparison, for example, to the extreme that a language that is strictly typed like Java, C# or C++ explicitly defines the data type when creating a symbol, which makes vulnerability detection easier than most things will be known, allowing reasoning and analysis with confidence.

There is also the case of the JavaScript language, where the variable declaration is dynamic but allows you to write using explicit types, getting the advantages of a strongly typed language. This makes the static analysis of vulnerabilities easier than dynamically modifying them.

3 A DSL to describe the properties of a Programming Language

As mentioned and demonstrated, there are aspects of programming languages that grammar metrics cannot measure or capture, but which have a huge impact on the support, particularly when it comes to a static analysis tool.

In this sense, a domain-specific language (DSL) was created with the goal of describing the features of a programming language that bring positive or negative impact to the moment of support. The Properties Language is used in programming languages to define the substance of linguistic features and paradigms.

This language was not created with the intention of acting as documentation for a specific language, but it does include in its design various characteristics and notions that allow for an assessment of its complexity.

The formal definition of the DSL to describe the Properties of a Language is shown in Listing 1.

Listing 1 Formal Definition of the Properties Language.

```
language -> header properties
header -> name (version)?
name -> ID
version -> VERSION
properties -> paradigms
    (SEMICOLON propertyDeclaration)?
    (SEMICOLON propertyIndentation)?
    (SEMICOLON propertyMemory)?
    (SEMICOLON propertySemiColon)?
    (SEMICOLON feature (SEMICOLON feature)*)?
    DOT
paradigms -> PARADIGM COLON paradigm (COMMA paradigm)*
paradigm -> nameParadigm (weight)?
weight -> DOLLAR NUM
propertyDeclaration -> DECLARATION COLON typeDeclaration (weight)?
typeDeclaration -> STATIC | DYNAMIC | BOTH
propertyIndentation -> INDENTATION COLON useIndentation (weight)?
useIndentation -> YES | NO
propertyMemory -> MEMORY COLON typeMemory (weight)?
typeMemory -> MANUAL | AUTOMATIC
```

```
propertySemiColon -> STRSEMICOLON COLON isMandatory (weight)?
isMandatory -> YES | NO | OPTIONAL
feature -> featureName COLON LPAR observations RPAR (weight)?
featureName  -> STR
observations -> observation (COMMA observation)*
observation -> STR

NUM -> '-'? [0-9]+
STR -> '"' ( ~('"'))* '"'
VERSION -> [0-9]+ (('.' [0-9]+)+)?
ID -> [a-zA-Z]+
```

The language concepts presented are quite explicit, in that the terminology used literally means that name. The phrases in this language are precisely the features of a programming language. Briefly, a reference about the name of the paradigms in which it can be any within a short list created and selected by the author.

Regarding the content of each list is one or more items within a paradigm, either a predefined feature or a free one that the user wants to describe. For each of these items, an associated weight (positive or negative) is optionally defined, thus translating the possibility of translating the impact of this element on the processing complexity.

▶ **Example 2.** *The following is an example of the description of the properties of the Python programming language, according to the DSL presented above.*

```
Python 3.10.4

PARADIGMS : object-oriented $3, procedural, functional,
    structured, reflective;

DECLARATION : dynamic ;

INDENTATION : yes $-1 ;

MEMORY : automatic $1 ;

SEMI-COLON : no ;

"Anonymous functions" : [
    "Implemented using lambda expressions;
        however, there may be only one expression in each body"
];
"String interpolation": [
    "The process of evaluating a string literal containing 1..*
        placeholders, yielding a result in which the placeholders are
            replaced with their corresponding values."
] $20;
"Triple-quoted": [
    "Beginning and ending with three single or double quote marks.",
    "May span multiple lines and function like documents in shells"
]
.
```

4 A Tool to determine Programming Language Complexity

Although there are some similar tools, such as the SynQ tool [9, 10], the SdfMetz, SdfCoverage [1, 2] and the gMetrics tool [3, 4] or even the GQE tool [5], that provide information about the complexity and quality of a grammar, this tool differs in its ultimate goal, to measure the complexity of a language rather than the grammar. However, these tools already demonstrate that grammar metrics are important for understanding the complexity of a grammar, therefore their implementation. Furthermore, this tool brings innovation, as it allows for an evaluation of linguistic properties through a new DSL.

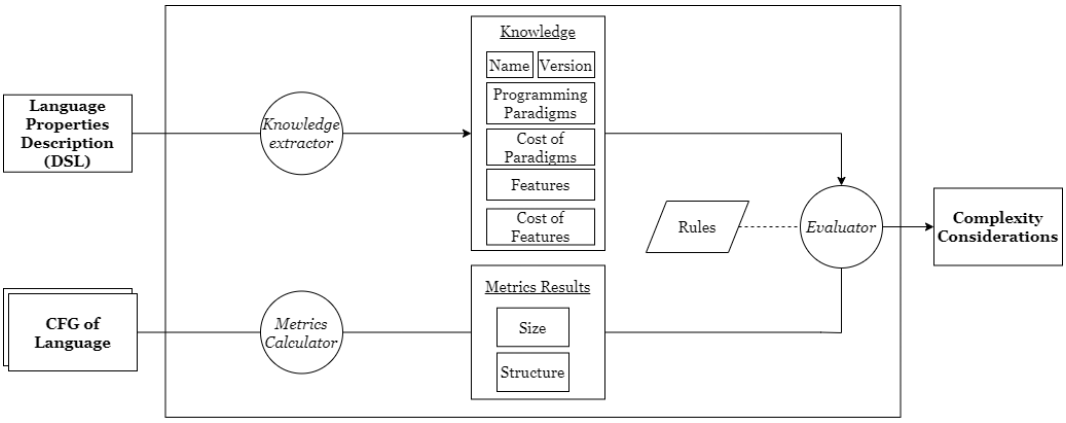

Figure 2 Language Complexity Evaluator, architecture.

The diagram of the grammatical representation of the tool under development is depicted in Figure 2. The tool, Language Complexity Evaluator (LCE), is an application that reads any ANTLR grammar (that is, any grammar written in the ANTLR metalanguage) and any text written in the metalanguage built by the author, generating values for each of the metrics under consideration and extracting knowledge from the properties provided. The user has the ability to analyse these values, as well as, observe a series of automatically generated considerations from a series of predefined rules. In this way, the user can easily predict whether the language is of low or high complexity.

Currently, the LCE tool is available as an open source project at `https://lce.di.uminho.pt/` and supports a variety of operations, the most notable of which are:

- Recognize a grammar in the ANTLR format and the description of features in the previously presented DSL format;
- Compute the size and structure metrics previously presented;
- Extract the knowledge from the features described using the DSL and their associated weights;
- Draw conclusions from the results obtained in the two previous phases.

4.1 Main Results

This subsection seeks to demonstrate the real application (using as a case study a well-known programming language) of the Language Complexity Evaluator tool, described above, in its present state publicly available.

The purpose is to demonstrate the advantages of utilizing this tool to estimate the time and effort needed to handle a new programming language in an SAST tool.

To do this, an actual grammar of a programming language will be utilized as a case study to demonstrate the various outcomes offered by the tool.

Python will be the programming language utilized as a case study. This programming is a general-purpose language, adaptable, and powerful. Because it is brief and easy to read, it is a good first language. Python can accomplish just about everything, from web development to machine learning.

The grammar that specifies the Python programming language, developed for ANTLR v4[2], and the description of the same programming language's features, shown in Example 2, are then considered as input files.

The outcome of the LCE tool's evaluation of the complexity of this language will be reported in two stages: first by the results of the metrics acquired from the grammar, and subsequently by knowledge extraction using the linguistic attributes supplied. Finally, some conclusions obtained from these two phases will be provided.

This tool effectively completes and automates the entire procedure, it also includes visual representations of the grammar structure and the Dependency Graph between symbols (DGS), as can be seen in the Figures 3 and 4.

Metrics Results

Tables 3 and 4 show the calculated values for each previously established grammatical metric in Tables 1 and 2.

Size Metrics

Table 3 Results of metrics for evaluating the Size of Context-Free Grammars.

Metric	Result
#P	122
#N	86
#T	89
#UP	6
RHS-Max	7
RHS	3.877
ALT	1.419
MCC	2.721

Structure Metrics

Table 4 Results of metrics for evaluating the Structure of Context-Free Grammars.

Metric	Result
#R	48
FanIn	5.5
FanOut	2.703
TIMP	43.917
CLEV	46.512
NSLEV	2
DEP	35

[2] The lexer and parser grammar can be found in the grammars-v4 repository of ANTLR.

- **Knowledge**

  Example 2 offers a brief overview of some of Python's linguistic features. The knowledge extraction received from LCE may be used to confirm the main characteristics of this language, such as:

  - It includes five programming paradigms, one of which, object-oriented, has a weight of three units. The tool also creates the following assignments: the functional paradigm received a value of three units, the procedural paradigm received a value of two, and the others received a value of one (reflective and structured). This takes the overall weight associated with the paradigms to 10 units.

  - As for the linguistic features themselves, seven were indicated, and four of them did not have an associated weight. The tool assigned a support weight to these features (declaration type, whether it enforces the use of semicolons, Triple-quoted feature, and the anonymous functions feature). Regarding the declaration type of variables and the use of semicolons, since both properties have a great impact on static vulnerability analysis, a weight of 9 and 10 units respectively was assigned. As for the other characteristics, a weight of 5 was assigned to each. This takes the overall weight attributed to the features to 49 units.

- **Complexity Considerations**

  The considerations, derived automatically by the tool, are actually helpful to the users. It gives complexity assumptions, letting the end user to reason about the language's support.

  In terms of grammar, LCE tool produces the following complexity issues:

  - A high size of the largest level indicates an uneven distribution of the non-terminals among grammatical levels;

  - The CFG is very large, in terms of productions and symbols, with many unitary productions and a low percentage of non-terminal recursive symbols, which contributes to its ease of understanding, manipulation, and maintenance.

  In terms of features, LCE tool produces the following complexity issues:

  - In a language that does not require the use of semicolons, it makes the support process in a static analysis tool extremely complex, because it involves additional effort to figure out where an instruction begins and ends.

  - This language by allowing a type declaration as dynamic does not require explicitly defining the data type when creating a symbol, making vulnerability detection more complex, as it is necessary to perform the compiler's work beforehand in order to know the information inferred by it, thus requiring more support effort.

  - This programming language requires its scopes to be delimited by their indentation, so the lexical analyser needs to do some work beforehand, since it needs to count the indentation, detect whether the block opens or closes, or whether it is the same, thus making the support process more time-consuming.

Figure 3 Visual representation of the grammar.

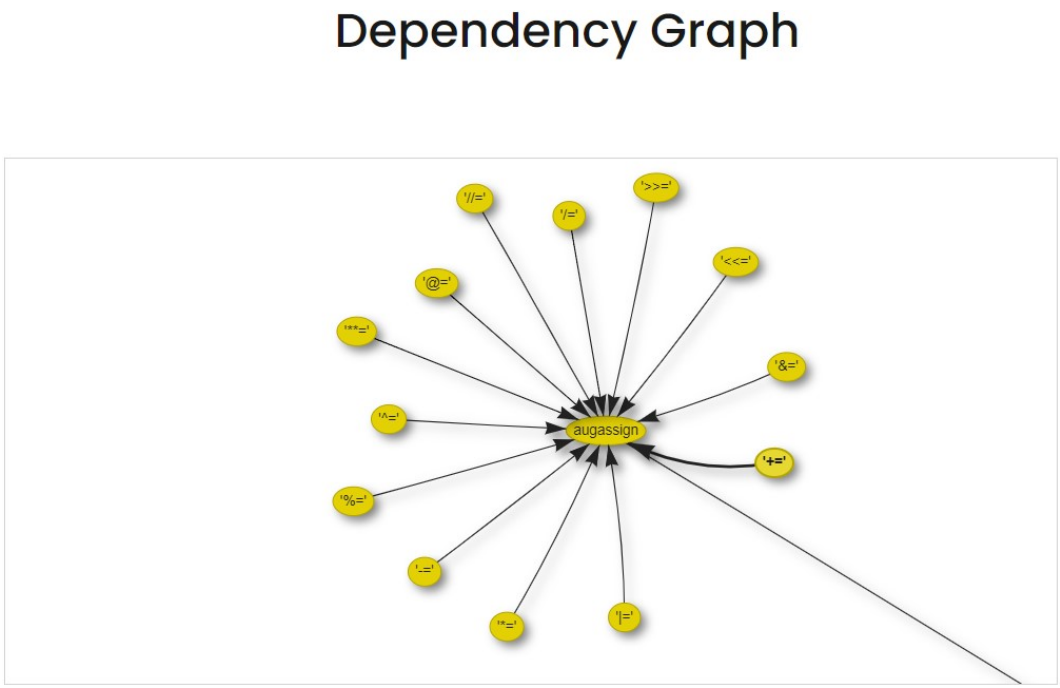

Figure 4 Visual representation of the DGS.

4.2 Predicting the effort to process a Programming Language automatically

In order to predict the effort to process a programming language automatically, MongoDB is used to store the previous languages already analysed for long-term storage, the information stored and used in the comparison are the metrics calculated, and the knowledge extracted.

MongoDB is a NoSQL document database that uses JSON-like documents to store data. MongoDB is used to store all language information.

This information comprises headers, paradigms, features, size metrics, and structure metrics. From the paradigms of a given language, a similarity search is performed with the languages already stored, and then the corresponding languages are presented with their respective values.

Following the same format as the previous part, a new case study will be introduced and analysed, this time to compare the complexity of programming languages.

▶ **Example 3.** *The following is an example of the description of the properties of the JavaScript programming language, according to the DSL presented above.*

```
JavaScript

PARADIGMS : event-driven, functional, imperative,
    procedural, object-oriented ;

DECLARATION : dynamic $7 ;

INDENTATION : no ;

SEMI-COLON : optional $9 ;

"Scoping" : [
    "Function scoping with 'var'",
    "Block scoping with the keywords 'let' and 'const'"
] $13;

"Weakly typed" : [
    "Means certain types are implicitly cast depending
        on the operation used"
] ;

"Anonymous function"  : [
    "A function definition that is not bound to an identifier.",
    "Anonymous functions are often arguments being passed to
        higher-order functions or used for constructing the result of
            a higher-order function that needs to return a function"
] ;
"Run-time environment" : [
    "typically relies on a run-time environment (e.g., a web browser)
        to provide objects and methods by which scripts can interact
            with the environment (e.g., a web page DOM)."
] ;
"First-class functions" : [
    "Function is considered to be an object",
    "A function may have properties and methods",
    "Created each time the outer function is invoked"
] ;

"Promises and Async/await" : [
    "Supports promises and Async/await for
        handling asynchronous operations."
]
.
```

The Example 3 shows a description of the properties of another programming language, JavaScript, according to the DSL presented above. With its grammar available in the same ANTLR grammars-v4 repository mentioned above, it is possible to make an automated and readable comparison between the two languages presented through the tool.

Figure 5 Comparison of results between Python and JavaScript.

Figure 5 compares the metrics collected, and the various paradigms introduced, along with their associated weights. Aside from this information, feature descriptions with weights for each language may be found independently.

In summary, the grammar used to express JavaScript is far bigger than Python's since it contains more productions and terminals; nevertheless, at the structural level, the JavaScript grammar is more like a graph in which the symbols are connected, whereas Python's is more like a tree. When it comes to attributes, the tool always depends on what the user describes, but based on these two samples, we can deduce that Python total was 49 and JavaScript total was 55. There are three common paradigms, and only the languages differ in the weight they place on the Object-Oriented Paradigm.

Therefore, the user may simply compare the material supplied by the LCE tool to the languages already supported, allowing them to forecast the work required to support a new programming language in its static analysis tool.

5 Conclusion

Along the paper, we examined ways to assist language support teams in improving their assessment of the time and effort necessary to deal with a programming language that is not yet supported in a Tool for Programming Languages processing, like Static Analysers.

The approach followed for assessing the language's complexity starts by first evaluating the difficulty of its underlying CFG. Factors causing the support process to take longer were found through the investigation of real case studies, particularly during the phases of language recognition and transformation to an AST. In this regard, in a second phase, a collection of linguistic traits is assessed in order to account for the aforementioned elements that also have an influence on language processing.

This research led to the development of a tool, available at `https://lce.di.uminho.pt/`, that allows for the evaluation of a language's difficulty based on a set of metrics to rate the complexity of its grammar with a set of properties. Furthermore, the tool analyses the difficulty of the new language as identified thus far with previously supported languages in order to forecast the effort required to process the new language.

5.1 Future Work

An important task, to be done in future work, is to improve the grammar considerations and the description of the provided features, using real grammars and linguistic properties of programming languages.

References

1 Tiago L. Alves and Joost Visser. Metrication of sdf grammars. Technical report, Universidade do Minho, 2005.

2 Tiago L. Alves and Joost Visser. A case study in grammar engineering. In *SLE*, 2008.

3 Julien Cervelle, Matej Crepinsek, Rémi Forax, Tomaz Kosar, Marjan Mernik, and Gilles Roussel. On defining quality based grammar metrics. *2009 International Multiconference on Computer Science and Information Technology*, pages 651–658, 2009.

4 Matej Crepinsek, Tomaz Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, and Gilles Roussel. On automata and language based grammar metrics. *Comput. Sci. Inf. Syst.*, 7:309–329, 2010.

5 João Cruz. Qge – An attribute grammar based system to assess grammars quality. Master's thesis, Universidade do Minho, December 2015.

6 Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Boston : PWS Publishing Company, 2 edition, 1997. ISBN: 0-534-95425-1.

7 Pedro Rangel Henriques. Brincando às linguagens com rigor: Engenharia gramatical. Technical report, Departamento de Informática da Escola de Engenharia da Universidade do Minho, November 2013.

8 Terence John Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009. ISBN: 978-1-934356-45-6.

9 J.F. Power and Brian A. Malloy. Metric-based analysis of context-free grammars. *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 171–178, 2000.

10 J.F. Power and Brian A. Malloy. A metrics suite for grammar-based software. *J. Softw. Maintenance Res. Pract.*, 16:405–426, 2004.