Upgrade of Lark Compiler Generator to Support Attribute Grammars

Daniel Faria[1]
ALGORITMI Research Centre/LASI – DI, University of Minho, Braga, Portugal

Tiago João Baptista
ALGORITMI Research Centre/LASI – DI, University of Minho, Braga, Portugal

Pedro Rangel Henriques
ALGORITMI Research Centre/LASI – DI, University of Minho, Braga, Portugal

Abstract

This document presents an initiative aimed at advancing parsing and language processing through the integration of Attribute Grammars (AG) into the Lark Compiler Generator, a flexible tool frequently used for these tasks. In order to achieve a successful integration of AGs into Lark, a study on the concept and example analysis of AG based specifications needed to be conducted. This provided an insight on their advantages in order to extend Lark with the ability to use AG, that will be presented in this article. With this project, named LarkAG, we aim at providing Lark with a conventional and well studied formalism to specify rigorously the static and dynamic semantics of programming languages, on top of its recognized syntactic analysis capabilities. Along the paper, LarkAG architecture, development and usage are also discussed. The DSL designed to provide a proper notation for attribute occurrences selection and for writing semantic rules and context conditions is also presented. This addition of AG support to Lark will enable the construction of compilers and language processors of greater reliability. It is not a major breakthrough since AGs based tools already exist and are well studied, but rather an extension that can bring great value to a modern framework such as Lark.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages

Keywords and phrases Formal Language Specification, Formal Language Processing, Attribute Grammars, Compiler Generation, Lark

Digital Object Identifier 10.4230/OASIcs.SLATE.2024.7

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

1 Introduction

Before diving into the specific details of Attribute Grammars (AG) and the Lark Processor Generator, it is important to establish the context in which these concepts exist. Understanding this background will provide valuable insight into the world of language processing.

Language and efficient communication are key facets of human interaction that shape our daily lives. The use of grammatical rules to formalize a language serves an important purpose, allowing us to transmit ideas precisely and without ambiguity. Furthermore, the systematic specification of languages via grammatical constructions is critical in the field of human-computer interaction, serving as the base structure for promoting unambiguous communication between men and machines. From the point of view of its automatic processing, for a grammar to be useful it should assure that a given symbol always has the same value wherever it is used, i.e. it must be a Context-Free Grammar (CFG).

[1] Corresponding author

Since their creation, CFG have played a pivotal role in compiler technology, speeding up and simplifying the process of implementing parsers. With the further exploration of the field of compiler generators, it becomes clear that a powerful parsing tool is indispensable for languages defined by a CFG. A plethora of parsers generator tools have been developed throughout the years to deal with the challenges of language processing and translation. These tools, which range from traditional parser generators to current parsing libraries, have been critical in facilitating programming language recognition and transformation.

From among them, Lark[2] stood out as a notable parsing generator, highlighted not only by its numerous capabilities but also for its user-friendly design. Lark's primary strength lies in its capability to parse any CFG, providing a versatile solution for any developer using CFG.

However, while CFG excel at defining the syntax of a language, it often falls short in capturing complex semantic relationships. This limitation motivated the birth of Attribute Grammars, fifty years ago. AG, as a formal mechanism for declaring and evaluating synthesized or inherited attributes associated with grammar symbols (NT or T), offer a structured and powerful approach to define rigorously semantic rules and enforce context conditions (which assures the semantic correctness of sentences). These facts prompt exploration of the integration of AG into the Lark framework.

By incorporating AG support into Lark, we address a longstanding need within the framework, enhancing its capabilities and extending its utility for more complex language processing tasks. This integration not only makes Lark more robust, but also aligns it with other advanced compiler tools that already leverage the power of Attribute Grammars.

This paper describes just that, LarkAG aims to improve Lark Compiler Generator's capabilities by overcoming the intrinsic restriction of CFG in capturing complex semantic relationships in the field of language processing. To accomplish such an objective it was necessary to design a DSL to be used by language engineers to write an entire AG with attributes of both types, evaluation rules, context conditions, and translation rules. This DSL and its processor (that translate the input AG into a Lark based Python compiler) will also be discussed.

After this introductory section, the paper is organized in seven more sections. Section 2 provides the necessary background on AG, presenting its formal definition and highlighting its theoretical and practical relevance.

To make the paper easily readable, Section 3 introduces Lark and summarizes its major features. This information will provide the background to understand the LarkAG extension. The fourth section will be dedicated to discuss and highlight the related works in the area of attribute grammar implementation. Section 5 will describe the system architecture used to address the stated problems and to develop LarkAG. In Section 6, LarkAG's domain-specific language is introduced, its grammar is presented, and an illustrative example of use is shown. Then, in Section 7, the current implementation progress and decision-making related to the development of LarkAG is discussed. Section 8 describes how a programmer can use this extension in their language engineering projects. At last, Section 9 closes the paper with some conclusions sorted out from the development of LarkAG language and processor, and with directions for future work.

[2] More details at `https://lark-parser.readthedocs.io/en/stable/`

2 Attribute Grammars, an overview

In this second section, the main concepts and relevance of Attribute Grammars (AG) will be presented. The main driving force of this project is the usefulness of Context-Free Grammars (CFG) in the world of language engineering, and the advantages brought about by their expansion using AGs. CFGs have been an important component of compiler technology since their creation in the 1960s. It turned the arduous, ad hoc, process of parser implementation into a much simpler and faster task [15].

AG formalism was first introduced as a means to express the semantics of computer languages, something that a CFG is incapable of [21]. Actually, an AG extends the primitive formalism CFG, providing a well-founded and formal mechanism for defining and handling attributes associated with the symbols of a grammar to express the language meaning [39].

An AG is composed of a context-free grammar (CFG), a set of attributes (A), a set of evaluation rules (ER), a set of context conditions (CC), and a set of translation rules (TR). This definition can be represented using the tuple:

$$\text{AG} = < \text{CFG, A, ER, CC, TR} >$$

The context-free grammar serves as the underlying structure, detailing the language's syntactic rules. Attributes serve as a powerful mechanism for attaching additional information to the symbols of a context-free grammar, allowing the specification of semantic properties, serving as individual computation units. These computations are typically simple and modular, allowing to implement sophisticated solutions for intricate programming challenges. They can also be examined, debugged, and maintained independently, making it easier to design and evolve programs [7]. There are two classes of attributes: inherited and synthesized. Inherited attributes (IA) are used to pass information down the tree, from parent nodes to children nodes. At a particular node in the syntax tree, synthesized attributes (SA) are calculated based on the attributes of their children, bringing information up from the leaves to the tree root [30].

Evaluation rules articulate the logic behind the computation of attributes. Attribute evaluation is a process that computes values of attribute instances within a syntax tree according to the evaluation rules. An attribute grammar is considered well-defined if the semantic rules [3] are such that for each program's syntax tree, the values of the attribute instances can be unambiguously computed by the evaluation process [15].

While the CFG productions ensure syntactic accuracy, the AG context conditions establish semantic validity. These predicates state, at the production level, the constraints that the attributes must follow in order for the input phrase to be meaningful [14]. Context conditions take the form of boolean-valued expressions over attribute occurrences [30].

After obtaining the meaning of the input text, translation rules are used to get the intended result [14].

Listing 1 shows a fragment of an AG aimed at defining a language for describing the students of a class and their grades. At the top of the listing, four declarations associate synthesized and inherited attributes with the non-terminal symbols `Start, Students` and `Student`. Then the AG fragment shown defines the attribute evaluation rules, only those associated with productions `p0` and `p1`, responsible for calculating the average grade of each student in the classroom. Those rules involve attributes `count` (that is incremented for each student recognized) and `grades` (a list collecting the name and average grade computed for

[3] This includes evaluation rules, context conditions and translation rules

each student). At the end of the listing, a context condition is associated with production `p1` in order to guarantee that there are no repeated student names. To impose that semantic constraint, the inherited attribute `nameList` associated with the symbol `Students` plays a crucial role, as it gathers the name of all the students previously parsed (on the left of the input sentence).

Listing 1 AG specification of Students Language.

```
SA(Start) = { count:int, grades:list(tuple(string,float)) }
SA(Students) = { count:int, grades:list(tuple(string,float)) }
IA(Students) = { nameList:list(string) }
SA(Student) = { avg:float, name:string }

p0: Start   -> 'Class' id Students '.'
ER(p0) = {
    Start.count = Students.count;
    Start.grades = Students.grades;
    Students.nameList = [];
}
p1: Students-> Students ';' Student
ER(p1) = {
    Students_1.count = Students_2.count + 1;
    Students_1.grades = [(Student.name, Student.avg)] ++ Students_2.
        ↪ grades;
    Students_2.nameList = Student.name ++ Students_1.nameList;
}
CC(p1) = {
    !(Student.name in Students_1.nameList);
}
...
```

The semantic rules related to a production cause local dependencies among the attributes associated with the symbols of that production, which must be understood in order to establish the right sequence of evaluation.

A convenient tool for describing a production and its semantic dependencies is a local dependency graph (called a DAG), where the nodes represent the occurrences of the production's attributes. In this graph, there is a direct arc between two nodes if one of them depends on the value of the other, i.e, the semantic rules which calculates one of them depends on the value of the other. To describe semantic dependencies of an entire syntax tree, rather than a single production, a (compound) dependency graph of the syntax tree needs to be defined [4]. The DAG can be constructed either statically – by the Compiler Generator, while it analyses the given AG – or dynamically – by the attribute evaluator – but in both approaches it is a complex and heavy process [1].

3 Lark - Concepts overview

In software development, parsing is the key to effective data processing. Since it bridges the gap between raw data and meaningful information that can be understood and used by computer programs, making it an essential component of many software systems. The efficiency and accuracy of parsing can have a direct impact on the functionality and performance of the software where it is included. As a result, choosing the correct parsing tool or library is critical for developers and can have a significant impact on project success.

This section will provide an overview of Lark's capabilities and significance in software development, while also delving deeper into its various features and inner workings.

Lark is a modern parsing library for Python that can parse any CFG. Parsers are inherently intricate and confusing and can be difficult to comprehend, write and use [37]. Its value in the field of parsing can be described as, but not only, a list of the following capabilities[4]:

- Offer multiple parsing algorithms, allowing the developer to choose the most suitable parsing technique for their specific tasks;
- Integrate lexical analysis and parsing into a single library, streamlining the process of language processing;
- Express grammars using an Extended Backus-Naur Form (EBNF) inspired notation, making it intuitive for developers to define parsing rules;
- Generate custom parse trees that can be adapted to match the structure of the grammar;
- Provide detailed error reporting and flexible error handling, helping developers in diagnosing and resolving parsing issues;
- Provide parse tree visualization, that can help users while writing custom visitors.

Lark's contribution to parsing is notable not only because of its capabilities, but also because of its usability and user-friendly design.

Lark is built upon various fundamental concepts that are key to understanding its functionality. These concepts include grammars, parse trees and parsing algorithms.

As mentioned earlier, grammars are the foundational element upon which the entire parsing process is built. In the context of Lark, grammars act as blueprints for how data in that language should be arranged and processed, establishing the language's syntax and structure.

This structure can be depicted in a hierarchical manner, using parse trees which serve as an intermediate representation of the parsed language and are constructed based on the input data and the defined grammar.

Lark builds these trees where each rule that is matched becomes a node in the tree, and its children are its matches, in the order of matching. In order to build parse trees, Lark employs parsing algorithms to analyse and understand the structure of the input data based on the defined grammars.

Lark implements three parsing algorithms: Earley, LALR(1), CYK[5], allowing developers to choose the one that best suits their needs. Parsing algorithms are collections of rules and procedures that govern how the library reads and interprets data.

Traditional parsing tools usually separate lexical analysis from syntactic analysis, but Lark streamlines the entire process, offering a unified solution for developers. In Lark, a single grammar is defined, one that includes both lexical and syntactic rules. This grammar is written using EBNF, which allows for the specification of both tokens and the language's general syntax, simplifying language processor implementation.

As mentioned before, Lark uses a combination of lexical analysis and syntactic parsing to transform input strings into parse trees and offers customization options for their automatic construction using a collection of grammar features and classes for traversing the parse trees [37]. With this flexibility of being able to choose the way the tree is visited, Lark allows programmers to choose the most suitable approach for their parsing and tree manipulation needs.

[4] Extracted from the original website at: `https://lark-parser.readthedocs.io/en/latest/index.html`

[5] CYK is a legacy parser

4 Related Work

As said in the Introduction, the general adoption by the compiler community, of Chomsky's formalism to define programming languages, the context free grammar, paved the way for the development of a wide range of Parser Generators. The most well known and worldwide used[6], is YACC [16], the generator developed, in parallel with the Lexical Analyser Generator Lex [26], to build the C programming language Compiler, being itself written in C. YACC was inspirational in the education and research fields, and nowadays with the leading of Python it was reimplemented as a Python library called PLY.

A similar path was followed in the attribute grammars field. After the seminal work of Knuth, introducing the attribute grammar concept in the early seventies, many researchers have developed AG-based Compiler (Language Processor) Generators (AGbCG, for short). The surveyed tools have a very similar generic architecture: accept as input an AG, and generate a lexical and syntactic analyser and a set of tree walkers to evaluate the attributes, check the context conditions, and produce the desired output.

One of the main differences among the tools lies in the AG class[7] supported – a lot of them accept the LAG class, which is the simplest to implement because attributes can be evaluated in just one visit top-down to the tree, from left to right; other tools cope with OAG class, a more generic one that allows for the adoption of the same visiting order, independent of the syntax tree, which can be determined statically; and a small number support the Non-circular AG class, in which case the order is determined dynamically being necessary to assure that the DAG is not circular.

Delta system [27], by Bernard Lorho, was one of the pioneer AGbCG, in the late seventies. Following the declarative character of AG, Delta was implemented in a functional language. It didn't have a long life, but it was the root for FNC/SNC [17] also developed at INRIA-Rocquencourt by Martin Jourdan, a recursive evaluator for Strongly Non-Circular Attribute Grammars. With the collaboration of Didier Parigot et al. that system was further developed and in 1990 they published the evolution, OLGA [18].

In parallel with Lorho's work in France, a team at Helsinki University developed another AGbCG, named HLP78 [34], which was improved in the following years by Koskimies et al. giving birth to HLP84 [23]. Koskimies and Paaki evolved that system and proposed, two years later, TOOLS [22, 24].

In the same decade, in Germany, Uwe Kastens developed the GAG system [20] that supports a large class of attribute grammars, called Ordered-AG [19].

Also, the Dutch Computer Science community was also very active. Indeed, researchers like Lex Augustein [3], van Deursen, Jan Heering, Paul Klint [12], Ralph Lammel, Eelco Visser, Joost Visser, among others, developed and published many notable contributions in the area of attribute based language specification and automatic generation of language processing tools. However, the work of Svierstra and his team at Utrecht deserves a special mention because they revisited the functional character of AG and the incremental evaluation approach[8]. They contribute with the concept of High-order AG [38], and based on that the group came out with LRC [25], an interesting system with function memorization to improve the incremental attribute evaluator [36].

[6] Both in the industrial and in the academic worlds.
[7] The AG class is defined according to the type of dependencies among the attributes.
[8] A very hot topic in the eighties.

In parallel, also in the early nineties, at Lund University/Sweden, Görel Hedin started working with AG and object-oriented programming languages building an AGbCG to support DOOR AG [10]. Ten years later, after many improvements and contributions to the area, she introduced JastAdd [11],a meta-compilation system that supports Reference Attribute Grammars (RAG) and is still in use[9].

Concerning the logic programming approach, two authors proposed systems to cope with any class of AG taking advantage of DCG and producing Prolog attribute evaluators: Bijan Arbab [2]; and Pedro Henriques [13].

A bit later than the pioneers mentioned so far, William Waite and his team, at Boulder-/Colorado, developed a nice AGbCG called Eli [9], a powerful, flexible and user-friendly compiler construction system. From the United States came one of the most notable AGbCG: ANTLR [33, 32], developed by Terence Parr, which is an evolution of PCCTS [31]. ANTLR is very flexible, robust and still in use nowadays; like YACC, it is a valuable tool in academy for research, teaching and in the software industry.

In the late nineties Marjan Mernik develop in Maribor/Slovenia another powerful and interesting tool for compiler generation and teaching called LISA [29].

Maybe one of the most recent AGbCG is Silver[10], an extensible attribute grammar specification language and system, developed in Minnesota/USA by Eric van Wyk[11] and colleagues [40] to investigate high modularity in language specification.

As stated above, in the late eighties there were so many publications on attribute grammars classification, applications, and evaluation approaches, as well as so many AGbCG, that Pierre Deransart et al. published three INRIA technical reports that became a LNCS survey collecting all this information [6]. In 1990, the same team organized WAGA [5], an international workshop to discuss *Attribute Grammars and their Applications*, which was attended by Knuth itself and most of the authors referred in the bibliography here cited.

5 LarkAG - Architecture

The success of any system depends not only on its functionalities, but also on the efficiency and coherence of its underlying architecture. This section presents the architecture of our developed system in order to provide insight on the key components and relationships that compose it. Throughout this section, while exploring the system's architecture, two layers will be analysed: the Generator Layer and the Processor Layer. The first contains components dedicated to the compilation and generation of other components, based on the attribute grammar, while the last focus on the processing of input phrases with the support of the previously generated components, in order to obtain the desired output. Figure 1 depicts these two layers and the main components of the developed system and their collaboration within said system, as well as the data exchanged from one component to the other.

[9] See more at `https://jastadd.cs.lth.se/web/`
[10] For all the details and many more references see `https://melt.cs.umn.edu/silver/`
[11] One of the authors still very active in the area, as can be seen in one of his most recent publications of 2023 [35].

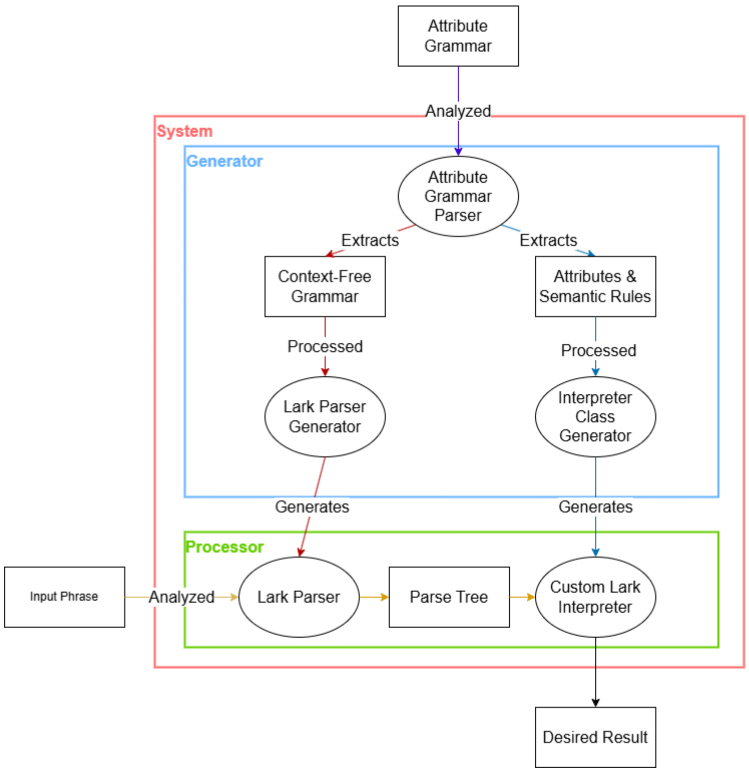

Figure 1 System Architecture.

The system accepts two inputs: the Attribute Grammar (AG) describing semantic rules and the input phrase to be processed. The output is generated with the semantics obtained from the AG, through a structured process that involves: AG parsing, translation, and dynamic module generation.

5.1 Generator Layer

At the heart of the proposed system lies the Generator Layer, designed to convert abstract attribute grammar specifications into concrete modules capable of semantic comprehension.

The first of these components to interact with the attribute grammar (which is an input from the user) is the Attribute Grammar Parser. This component extracts the essential information related to attributes and their semantic rules, as well the underlying Context-Free Grammar (CFG), by parsing the input AG, created using the DSL mentioned in 6.

The Lark Parser Generator is crucial in designing the syntactic interpretation of the user-defined language within this system, taking the CFG extracted from the AG and producing a parsing engine, tailored to the grammar's syntax.

The last component of this layer is the one responsible for separating the whole system from an ordinary Lark Parser Generator, the Interpreter Class Generator. This generator delves into the intricacies of semantics, generating a custom interpreter class built to operate over the parse trees, providing meaning to them. Taking the attributes and semantic rules extracted by the AG Parser, this component will dynamically generate an Interpreter Class [37] and its methods, enabling the calculation of attributes, evaluation of context conditions and the application of translation rules.

5.2 Processor Layer

The Processor Layer encapsulates the dynamic execution of language interpretation within our architecture. This layer, which consists of the Lark Parser and the custom Interpreter Class (built by the Interpreter Class Generator) converts parsed input phrases into the desired output.

The Lark Parser's primary task is to parse the input phrases based on the CFG extracted by the generator layer, and by doing so, it constructs a parse tree that reflects the hierarchical relationships among the various language elements.

The Custom Lark Interpreter, generated by the Interpreter Class Generator, assumes a crucial role, serving to close the gap that separates syntax and semantics. As the executor of semantic analysis, its task is to digest the parse tree (built by the Lark Parser) navigating the hierarchical structure to introduce meaning into the syntactic relationships between the language elements.

6 LarkAG - Domain-Specific Language

Many computer languages are domain specific rather than general purpose. Domain-Specific Languages (DSLs) sacrifice generality in favour of heightened expressiveness within a specific domain. Through the provision of notations and constructs specifically designed for a particular application domain, DSLs deliver significant improvements in expressiveness and user-friendliness compared to General-Purpose Languages (GPLs) in the given domain. This leads to increased productivity and decreased maintenance costs [28].

In this section, the DSL that provides the user with a means of expressing semantic rules and attributes in their language specifications is presented. The scope of this DSL is limited to the specification of the attributes, evaluation rules, context conditions, and translation rules – the quintessential elements of an attribute grammar. The definition of the syntax of the grammar will be done using the EBNF, just like in Lark - Concepts overview The DSL uses a combination of keywords, operators, and special symbols to express relationships and behaviours within the language semantics that have been described. The section below will provide an overview of the key elements of the DSL.

Non-terminal symbols in the DSL are represented using identifiers, adhering to the pattern: `[A-Za-z][A-Za-z0-9]*`, while terminal symbols are represented by encasing them with quotation marks.

Expressing attributes, inside a semantic operation, involves using an object-oriented notation using a syntax akin to: `<non-terminal>.<attribute>`. This notation means that attributes are associated with specific non-terminals, adopting a structure reminiscent of object-oriented programming where the non-terminal serves as an object and the attribute as a property of said object, this will be the notation used inside the definition blocks. When there's multiple non-terminals with the same name, inside a production, in order to disambiguate which non-terminal attributes are being accessed, an index of the non-terminal being accessed should be provided, counting from left to right using one-based indexing. This will follow the syntax: `<non-terminal>[index].<attribute>`.

Each of the composing element's definition follows a structured format, encased within a block. This organized approach is critical for preserving clarity and coherence when expressing various aspects of the attribute grammar. Each component is enclosed within a block that takes the following form:

```
<Type>(<production|non-terminal>)?{
    <Definition>;
    <Definition>; ...            }
```

When defining the attributes related to a certain non-terminal symbol, the type of attributes[12], and which non-terminal symbol it refers to need to be specified. The syntax of an attribute declaration block is as follows:

```
IA(<non-terminal symbol>){...} //Inherited Attributes
SA(<non-terminal symbol>){...} //Synthesized Attributes
```

Within an attribute definition block, the DSL allows for the explicit specification of attributes associated with a certain non-terminal symbol, requiring the explicit declaration of its name and type, where the type must be one of the Python programming language's built-in types [8]. The syntax for defining an attribute is as follows: `<attribute_name>:<attribute_type>`

To define the semantic rules[13], the type of the semantics being defined needs to be specified. For each expansion inside each production, the semantic operations need to be declared right after defining that expansion. The syntax of a semantic rule declaration block is:

```
ER{...} //Evaluation Rules
CC{...} //Context Condition
TR{...} //Translation Rules
```

The evaluation rule block requires that all statements express assignments, as indicated by the necessary use of the equals sign (=). This means that each statement, according to the syntax, must include an assignment operation, emphasizing the assignment-based nature inherent in the creation of evaluation rules. The syntax of an evaluation rule statement is: `<attribute> = <expression>`.

In order to define a context condition, inside a context condition definition block, it's necessary for the condition to function as a logic statement, so that they can be verified to be true, in order for the program to function.

When defining a translation rule definition block, each statement needs to encapsulate a function call, either being a default Python function or explicitly defined by the user. This function takes attributes as parameters and constructs the desired output. The syntax is as follows: `<pre_defined_function>(<attribute1>,<attribute2>,...)`

The following example showcases the portion of the attribute grammar presented earlier in AG specification of Students Language, using the newly introduced DSL. This example shows the application of the DSL within the context of the previously shown attribute grammar, elucidating its practical implementation and functionality. The DSL, resembles a more mathematical notation, retaining its expressiveness and formal accuracy while incorporating a "Pythonic" syntax to provide more familiarity and ease of use. This mix of mathematical principles and "Pythonic" expressions allow users to express attribute grammars in a simple and natural way, increasing efficiency and clarity during the writing process.

Listing 2 DSL Example.

```
SA(start) { count:int, grades:list[tuple] }
SA(students) { count:int, grades:list[tuple] }
IA(students) { name_list:list[str] }
SA(student) { avg:float, name:str }

start: "Class" ID students "."
```

[12] SA for synthesized attributes, IA for inherited attributes
[13] evaluation rules, context conditions and translation rules

```
ER {
    start.count = students.count;
    start.grades = students.grades;
    students.name_list = [];
}

students: students ";" student
ER {
    students[1].count = students[2].count + 1;
    students[1].grades = students[2].grades + [(student.name, student
        ↪ .avg)];
    students[2].name_list = students[1].name_list + [student.name];
}
CC {
    student.name not in students[1].name_list;
}
...
```

7 LarkAG, Development

The development process of the Lark extension involved a systematic approach to building and enhancing the capabilities of the Lark parser generator.

The project began with the design of the system architecture, specified in LarkAG - Architecture. Key design decisions were made to ensure the effectiveness and flexibility of the LarkAG extension. These decisions included the adoption of a modular architecture to facilitate extensibility and maintainability, the use of "Pythonic" design patterns to enhance readability and usability, the usage of expressive and meaningful intermediary data structures, to allow the various modules to communicate flawlessly.

As a part of designing the system architecture, the five main components of our system were identified: the AG Parser, the Lark Parser Generator, the Interpreter Class Generator, the Lark Parser, and the Custom Interpreter Class. Out of these components, the AG Parser and the Interpreter Class Generator were designed and developed from scratch to meet the specific requirements of the project.

The AG Parser employs the Lark Parser Generator, which is configured with a grammar tailored to the Domain-Specific Language (DSL) defined in LarkAG - Domain-Specific Language. This grammar enables the creation of a specialized parser capable of parsing the AG input provided by the user.

Upon parsing, the Lark Parser generates a Parse Tree representing the syntactic structure of the input. This tree is then fed into a Transformer that traverses it, extracting essential information such as the Context-Free Grammar (CFG) and the semantic information comprising attributes and semantic rules. The CFG is obtained in string format, representing the structural elements of the grammar. Simultaneously, the Transformer constructs a dictionary encapsulating the semantic information extracted from the Parse Tree, that follows a hierarchical structure akin to the Parse Tree, albeit in a simplified form.

This streamlined representation facilitates efficient storage and manipulation of semantic information, laying the groundwork for the subsequent stage of generating the Interpreter Class inside the Interpreter Class Generator.

The Interpreter Class Generator takes the previously mentioned data structure in order to dynamically generate a Custom Interpreter Class, which encapsulates and executes the semantic rules defined within the Attribute Grammar.

For each expansion within every production of the Attribute Grammar, the Interpreter Class Generator creates dedicated methods responsible for executing the semantic operations specified by the user. This method generation process follows a systematic procedure, encompassing the following key steps:

- Visitation of Children Nodes: The generator determines which child nodes of the current node need to be visited. For each of these child nodes, an operation responsible for its visitation is added to the list of operations.
- Topological Ordering: A topological order for all semantic operations associated with the expansion is established. This order ensures that semantic operations are executed in a logical sequence.
- Variable Replacement: Replace all occurrences of attributes and reserved words for the corresponding variables.

By following these steps, the Interpreter Class Generator enables developers to translate abstract semantic rules into executable code, enabling robust and efficient semantic analysis within the context of the Lark extension.

During the development of the LarkAG extension, various challenges arose. One such challenge was managing the dependencies between semantic operations in the AG. Ensuring that these operations are executed in the correct order is crucial to semantic analysis. To address this challenge, we implemented a depth-first search (DFS)-based algorithm for topological ordering. The topological ordering function calculates the order of semantic operations by traversing the dependency graph of a given expansion. This way, we ensure that each semantic operation is executed only after all its dependencies have been resolved. LarkAG may face challenges in handling exceptionally complex grammars with numerous interdependencies. Another inherent limitation of the topological sorting algorithm is its inability to handle circular dependencies within attribute grammars. If the attribute grammar contains circular dependencies, the algorithm will fail to determine a correct topological ordering, leading to potential runtime errors or infinite loops. Detecting and resolving such circularities is a non-trivial problem and remains an area for further development.

Another challenge was efficiently and correctly translating the attribute calls present in the semantic expressions into variables that Lark can use for attribute evaluation. We developed a helper function to dynamically translate attribute calls into tree references. This function processes a node iteratively and organizes its child nodes into a dictionary. Each entry in the dictionary matches a given non-terminal and index to its corresponding child node reference. This organized structure and its dynamic nature enable efficient access to child nodes during attribute evaluation, and the generated code to be used in rules with highly variable number of child nodes. This solution comes with performance trade-offs, due to its dynamic nature. The call to the helper function will add an overhead to each method that evaluates the semantic expression. This overhead can prove significant in the case of complex grammars with lots of rules.

8 LarkAG - Usage

The LarkAG extension offers developers a powerful tool for seamlessly integrating attribute grammars (AG) into their projects. This section provides a concise overview of LarkAG's core features and functionalities which enable such.

In order to use the LarkAG extension in their projects, developers can simply install it using pip, a popular Python package manager. Once installed, they can import it into their Python scripts and start leveraging its capabilities for enhanced semantic analysis.

When developing a project in which LarkAG is employed, developers have the choice between using it as a complete and self-contained system or using only the generator layer.

The first approach involves using a single class which encapsulates both the generator and processor layers of LarkAG. This class serves as a unified interface for semantic analysis, taking as input the attribute grammar and the source code to be analysed. Internally, the class utilizes the generator layer to parse the attribute grammar and generate the processor layer which will be responsible for processing the input source code, facilitating semantic analysis and producing the desired output within the project environment.

The second method involves leveraging LarkAG's generator layer exclusively. In this context, developers can use LarkAG with the sole objective of producing source code based on the provided attribute grammar. After generating the source code, developers can manually integrate it into their projects, incorporating it as a component of their comprehensive solution for semantic analysis. This approach allows integrating semantic analysis into their projects in a way that best suits their specific requirements and workflows.

The following code exemplifies the product yielded by the Generator Layer when provided with the attribute grammar presented in DSL Example. This output encompasses the Context-Free Grammar (CFG), annotated with aliases aimed at helping the generated Interpreter distinguish between each production's expansions. Additionally, the output contains the Custom Interpreter Class, which is designed to incorporate and execute the semantic operations expressed within the attribute grammar given as input.

Listing 3 Generated Code.

```
start : "Class" ID students "." -> start_0
students : students ";" student -> students_0 | student -> students_1
student : name "(" grades ")" -> student_0
grades : grades "," NUM -> grades_0 | NUM -> grades_1
...
%import common.WS
%ignore WS

class MyInterpreter(Interpreter):
    ...
    def start_0(self, node):
        pointers = self.__helper__(node)
        pointers['students'][0].name_list = []
        self.visit(pointers['students'][0])
        node.count = pointers['students'][0].count
        node.grades = pointers['students'][0].grades

    def students_0(self, node):
        pointers = self.__helper__(node)
        self.visit(pointers['student'][0])
        pointers['students'][1].name_list = node.name_list + [
            ↪ pointers['student'][0].name]
        self.visit(pointers['students'][1])
        node.count = pointers['students'][1].count + 1
        node.grades = pointers['students'][1].grades + [(pointers['
            ↪ student'][0].name, pointers['student'][0].avg)]
        if(not(pointers['student'][0].name not in node.name_list)):
            raise Exception('Semantic Error')
    ...
"""
```

9 Conclusion

In this section, we summarize the outcomes reached so far.

The concept of CFG and more importantly AG was presented and explained, by looking into formal definitions and writing examples. After introducing Lark framework and surveying the related work on Attribute Grammar based Compiler Generators (AGbCG), it was possible to motivate our project, LarkAG, which aims to be a modern and improved solution carrying on the approach for language semantics formalization into the Python world.

Another key concept worked and discussed was the DSL, which intends to be simple, clear to write/read, and at the same time maintain the specification expressiveness that characterizes AG.

Also, it is important to highlight the contributions of the architecture that lays at the foundations of AG in Lark, the division in two separate layers. The first one is dedicated to generation, and is responsible for recognizing and extracting the information from the AG written in the LarkAG DSL to then produce the Lark Parser and the Custom Lark Interpreter. This clear division permitted to have a structured and well-thought-out approach going into the development phase, which allowed LarkAG to be efficient, easier to understand and extend (maintain). The second one is the product of the first, and converts input phrases into the desired output, using both the Lark Parser (to construct the parse tree) and the Custom Lark Interpreter (to traverse the parse tree).

As it stands, LarkAG allows building an end-to-end parser based on CFG enriched with attributes, allowing users to generate a Lark Interpreter that does exactly what is intended according to the specified context conditions and translation rules defined in the AG. It gives more formality and assurance to the process of building a compiler. Furthermore, it saves the process of creating the Interpreter class by hand, and avoids the possible code errors/misbehaviour usually introduced during that process. It also shortens the development process by avoiding code clean-ups and bug corrections.

The next planned step is to test LarkAG using actual users and real projects. In order to achieve this, it will be created a web interface incorporating: an editor to write the specification in the LarkAG DSL; a compiler which allows users to generate the desired language processor; and a run functionality to test the processor. This web platform will allow collecting information about how the users actually interact with the new tool and might also help to find bugs/inconsistencies in LarkAG.

References

1 Henk Alblas. Attribute evaluation methods. In *International Summer School on Attribute Grammars, Applications, and Systems*, pages 48–113. Springer, 1991. doi:10.1007/3-540-54572-7_3.

2 Bijan Arbab. Compiling circular attribute grammars into prolog. *Journal of Research and Development*, 30(3):294–309, May 1986. doi:10.1147/RD.303.0294.

3 Lex Augsteijn. The elegant compiler generator. In P. Deransart and M. Jourdan, editors, *WAGA-90*, pages 238–254. Springer-Verlag, September 1990. LNCS 461.

4 Rina Cohen and Eli Harry. Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 121–134, New York, NY, USA, 1979. Association for Computing Machinery. doi:10.1145/567752.567764.

5 P. Deransart and M. Jourdan, editors. *Attribute Grammars and their Applications*. INRIA, Springer-Verlag, September 1990. Lecture Notes in Computer Science, nu. 461.

6 Pierre Deransart, Martin Jourdan, and Bernard Lorho. Attribute grammars: Main results, existing systems and bibliography. In *LNCS 341*. Springer-Verlag, 1988.

7 João Paulo Fernandes, Pedro Martins, Alberto Pardo, João Saraiva, and Marcos Viera. Memoized zipper-based attribute grammars. In *Programming Languages: 20th Brazilian Symposium, SBLP 2016, Maringá, Brazil, September 22-23, 2016, Proceedings 20*, pages 46–61. Springer, 2016. `doi:10.1007/978-3-319-45279-1_4`.

8 Python Software Foundation. Python language reference, version 3.12, 2024. URL: `https://docs.python.org/3.12/`.

9 R. Gray, V. Heuring, S. Kram, A. Sloane, and W. Waite. Eli: A complete, flexible compiler construction system. Research report, Univ. of Colorado at Boulder, October 1990.

10 Görel Hedin. Incremental static semantic analysis for object-oriented languages using Door Attribute Grammars. Research Report LU-CS-TR:91-79, Dep. of Computer Science, Lund Institute of Technology, August 1991.

11 Görel Hedin and Eva Magnusson. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003. Special Issue on Language Descriptions, Tools and Applications (L DTA'01). `doi:10.1016/S0167-6423(02)00109-0`.

12 Jan Heering and Paul Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, 35:2000, 1999.

13 Pedro R. Henriques. A semantic evaluator generating system in PROLOG. In P. Deransart, B. Lorho, and J. Maluszynski, editors, *Programming Languages Implementation and Logic Programming*, pages 201–218. Springer-Verlag, May 1988. LNCS 348.

14 Pedro Rangel Henriques. Brincando às Linguagens com Rigor: Engenharia Gramatical. Habilitation in Computer Science (Technical Report), Dep. de Informática, Escola de Engenharia da Universidade do Minho, October 2011. Habilitation monography presented and discussed in a public session held in April 2012 at UM/Braga.

15 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

16 Stephen C. Johnson. YACC Yet Another Compiler Compiler. Computing Science Technical Report CSTR32, Bell Laboratories – Murray Hill, New Jersey, 1975.

17 Martin Jourdan. An efficient recursive evaluator for strongly non-circular attribute grammars. Rapport de Recherche 235, INRIA, Rocquencourt, October 1983.

18 Martin Jourdan, Carole Le Bellec, and Didier Parigot. The OLGA Attribute Grammar Description Language: Design, implementation and evaluation. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, pages 222–237. Springer-Verlag, September 1990. LNCS 461. `doi:10.1007/3-540-53101-7_16`.

19 Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980. `doi:10.1007/BF00288644`.

20 Uwe Kastens, B. Hutt, and E. Zimmermann. GAG: A practical compiler generator. In *LNCS 141*. Springer-Verlag, 1982.

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

22 Kai Koskimies, T. Elomaa, T. Lehtonen, and J. Paakki. Tools/hlp84 report and user manual. Research Report A-1988-2, Univ. of Helsinki, Depart. of Computer Science, 1988.

23 Kai Koskimies, O. Nurmi, J. Paakki, and S. Sippu. The design of the language processor generator hlp84. Techn. Report A-1986-4, Univ. of Helsinki, Depart. of Computer Science, 1986.

24 Kai Koskimies and Jukka Paakki. Tools: An unifying approach to object-oriented language interpretation. Research report, Univ. of Helsinki, Depart. of Computer Science, 1987. (draft).

25 Matthijs Kuiper and João Saraiva. Lrc — a generator for incremental language-oriented tools. In Kai Koskimies, editor, *Compiler Construction*, pages 298–301, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. `doi:10.1007/BFB0026440`.

26 M. E. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. Computing Science Technical Report 39, Bell Laboratories – Murray Hill, New Jersey, 1975.

27 Bernard Lorho. Semantics attributes processing in the system DELTA. In A. Ershov and C.H.A. Koster, editors, *Methods of Algorithmic Language Implementation*, pages 21–40. Springer-Verlag, 1977. LNCS 47.

28 Marjan Mernik, Jan Heering, and Anthony Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–, December 2005. `doi:10.1145/1118890.1118892`.

29 Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/interpreter generator system LISA. In *IEEE Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000.

30 Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27(2):196–255, 1995. `doi:10.1145/210376.197409`.

31 T. J. Parr, H. G. Dietz, and W. E. Cohen. PCCTS reference manual: version 1.00. *ACM SIGPLAN Notices*, 27(2):88–165, February 1992. `doi:10.130973.130980`.

32 Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2nd edition edition, January 2013.

33 Terence Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995. `doi:10.1002/SPE.4380250705`.

34 K.J. Räihä, M. Saarinen, M. Sarjakoski, S. Sippu, E. Soisalon-Soininen, and M. Tienari. Revised report on the compiler writing system hlp78. Techn. Report A-1983-1, Univ. of Helsinki, Depart. of Computer Science, 1983.

35 N. Ringo, L. Kramer, and E. Van Wyk. Nanopass Attribute Grammars. In J. Saraiva, T. Degueule, and E. Scott, editors, *16th ACM SIGPLAN International Conference on Software Language Engineering (SLE2023, Co-located with SPLASH2023)*, pages 70–83. Association for Computing Machinery, Inc., 2023. `doi:10.1145/3623476.3623514`.

36 João Saraiva, S. Swierstra, and Matthijs Kuiper. Functional Incremental Attribute Evaluation. In David A. Watt, editor, *Compiler Construction*, pages 279–294, Berlin, Heidelberg, March 2000. Springer Berlin Heidelberg. `doi:10.1007/3-540-46423-9_19`.

37 Erez Shinan. Lark documentation, 2020. URL: `https://lark-parser.readthedocs.io/en/latest/index.html`.

38 S.D. Swierstra and H.H. Vogt. Higher Order Attribute Grammars: a Merge between Functional and Object Oriented Programming. Research Report RUU-CS-90-12, Dep. of Computer Science / Utrecht Univ., March 1990.

39 Martti Tienari. On the definition of an attribute grammar. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 408–414. Springer Berlin Heidelberg, 1980. `doi:10.1007/3-540-10250-7_31`.

40 Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming*, 75((1-2)):39–54, January 2010. `doi:10.1016/J.SCICO.2009.07.004`.