

Comparison of Leading Language Parsers – ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, Bison

Afshan Latif¹, Farooque Azam¹, Muhammad Waseem Anwar², and Amina Zafar¹

¹Department of Computer and Software Engineering, College of Electrical and Mechanical Engineering, National University of Sciences and Technology (NUST), Islamabad, Pakistan

²Department of Innovation, Design, and Engineering, Malardalen University, Vasteras, Sweden

alatif.cse20ceme@student.nust.edu.pk, farooq@ceme.nust.edu.pk,
muhammad.waseem.anwar@mdu.se, azafar.cse21ceme@student.nust.edu.pk

Abstract—Software engineering applications in domains like embedded systems and health care have increased exponentially during the last few years. Developing, analyzing, and customization of languages is one of the core software engineering aspects. This usually involves lexical, syntactical, and semantic operations, technically termed parsing. For this, several parsers have been introduced in state-of-the-art. However, due to diverse features, selecting a parser for a particular operation during software engineering applications is always problematic. In this article, we identified six leading parsers (i.e., ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, and Bison) from the state-of-the-art. Furthermore, we also identified significant parser features to perform meaningful comparative analysis. Results indicate that ANTLR and JavaCC provide enhanced parsing features, such as the parsing algorithm and the extended grammar notation. However, JavaCC is suitable for simple grammar definition, whereas ANTLR allows specifying complex grammar with multiple alternative paths. The findings of this article are highly beneficial for researchers and practitioners while selecting the right parser to perform specific software engineering tasks.

Keywords—*antlr, javacc, sablecc, tree-sitter, yacc, bison, parser comparison*

I. INTRODUCTION

In recent years, software engineering has witnessed a remarkable surge in the utilization of software applications across various domains, including the rapidly evolving field of embedded systems and healthcare, etc. Within software engineering, one of the core aspects is developing, analyzing, and customizing programming languages [1]. These activities often involve parsing. Parsing refers to analyzing the source code's structure based on a given set of rules or grammar [2]. Parsers are created manually or by parser generators. They use a lexer, also known as a lexical analyzer, to identify the terminal symbols of a specified language, breaking it down into smaller units such as tokens or abstract syntax trees (ASTs) and verifying that they conform to the language's syntax and semantics [3]. Parsing involves several sub-tasks performed sequentially or in parallel, depending on the algorithm. The first three tasks in Fig. 1 represent the main sub-tasks of parsing: lexical analysis, syntactic analysis, and semantic analysis [4].

Lexical Analysis: Lexical Analysis involves tokenizing the input source code into a sequence of lexical units or tokens, such as keywords, identifiers, literals, and punctuation symbols.

Usually, a lexer or scanner completes this step by reading the input source code character by character and producing a stream of tokens.

Syntactic Analysis: Syntactic Analysis analyzes the sequence of tokens to determine whether it conforms to a specified grammar, typically context-free grammar (CFG). The output of this syntactic analysis is a parse tree or an abstract syntax tree (AST), which represents the hierarchical structure of the input source code.

Semantic Analysis: Semantic analysis analyzes the meaning of the input source code beyond its syntax. This includes type checking, name resolution, and other checks that ensure the correctness of the code. The output of this semantic analysis is an annotated syntax tree, which represents the meaning of the input source code.

Parsing is typically done through two different techniques: top-down and bottom-up parsing [5]. Top-down parsing generally is more efficient but requires well-defined grammar. Bottom-up parsing, on the other hand, is more flexible and can handle a broader range of grammar, but it is usually complex [6].

Parsing facilitates numerous software engineering tasks such as program comprehension, code generation, and software verification [7]. Given the significance of parsing in software engineering, a wide range of parsers have been introduced in the state-of-the-art. However, selecting an appropriate parser for a specific software engineering task remains challenging due to different parsing tools' various features and capabilities. Choosing the right parser is crucial for the desired application, as an inappropriate parser will decrease performance and productivity. Researchers and practitioners often face the dilemma of choosing the most suitable parser to perform software engineering tasks effectively [8].

To address this challenge, this article aims to comprehensively analyze six leading parsers: ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, and Bison. These parsers were selected based on their prominence and widespread adoption in the existing literature. By focusing on these well-established parsers, we form a solid foundation for evaluating their performance and capabilities within the context of software engineering applications.

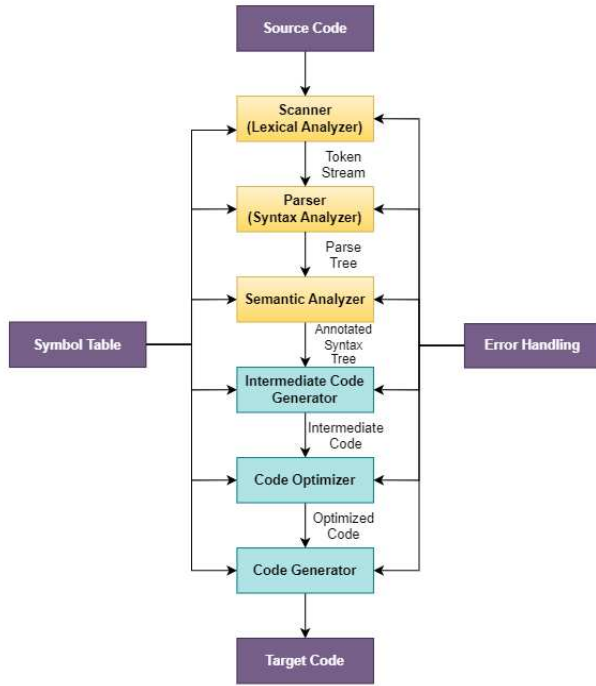


Fig. 1. Parsing Big Picture

A set of key features that significantly impact the parsing process have been identified to enable an effective comparison. These features are grouped into three major categories: parser, grammar, and editor specifications.

Considering these features, we provide a comprehensive and meaningful analysis of the leading parsers. Our study revealed that the parsing algorithm, input grammar notation, and the support for semantic predicates play a vital role in the overall performance of the parser. ANTLR, JavaCC, and SableCC support semantic predicates and specify their grammar through EBNF notation, allowing for more expressive grammar specifications through additional constructs. SableCC, a bottom-up parser, is complicated to implement. On the other hand, ANTLR and JavaCC are easy to implement, but JavaCC is more suitable for simple grammar, whereas ANTLR offers more complex grammar specifications with multiple feasible paths.

The findings of our study will assist researchers and practitioners in making informed decisions when selecting a parser for specific software engineering tasks.

The remaining paper is organized as follows: Section II overviews the literature relevant to our work. A comprehensive analysis of ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, and Bison based on different features is presented in Section III. The discussion on comparison with some key arguments and the direction for future work is provided in Section IV. Finally, we conclude our work in Section V.

II. RELATED WORK

To help the developers select the right tool, Cruz et al. [9] presented a comprehensive analysis of three compiler generators, including YACC, LISA, and ANTLR-3. For this, they introduced a Domain Specific Language (DSL) called

Lavanda, which computes the laundry bags a company sends to wash daily. Lavanda was implemented as an interpreter with the selected tools. The semantics of Lavanda was specified in the grammar and translated to all three tools. They evaluated these tools by analyzing various features, grouped into three categories: language specification, parser generation, and code generation. They concluded that YACC is the poorest, and LISA and ANTLR-3 are powerfully effective tools according to their defined comparison criteria.

Pehlivan, H. et al. [10] defined a programming language to support mathematical function structures and developed an interpreter to evaluate its source code. An LL(k) grammar defines the language's syntax, written in BNF notation. Each of the interpreter's fundamental components, such as the parser, semantic controller, and code evaluator, performs a distinct code interpretation. JavaCC is utilized to create the LL(k) parser for syntactic language analysis. Other components use the abstract syntax tree generated by JavaCC. Several mathematical algorithms, each of which performs calculations on a unique sequence of integers, are coded and read as code samples to demonstrate the language's capabilities.

Anwar, M. W. et al. [11] presented a blended modeling framework for various design and verification standards. This framework can automatically generate graphical and textual notations based on the provided meta-model and DSL grammar. Furthermore, a mapping editor is provided so that input from subject matter experts can be used to ensure proper mapping among graphical and textual notations. Implementing EBNF grammar in the JavaCC tool makes it easy to switch among different notations. This blended framework is developed in Sirius. The PSS use case demonstrates the feasibility of the proposed framework. In particular, the PSS textual and graphical notation is developed as part of a blended modeling environment for the specification of test intentions. Early findings are promising, demonstrating that the suggested framework can combine modeling standards from different domains.

Chen and Pager [12] empirically evaluated the frequently utilized LR (1) parsing algorithms. They used 17 simple and 13 programming language grammar. They implemented them with the Hyacc based on Knuth canonical algorithm LR (1), Pager's practical general method LR (1), lane-tracing algorithm LR (1), unit production elimination algorithm and its extension, and the edge pushing algorithm LR (k), and with the Bison based on the LALR (1) algorithm. They compared these algorithms based on performance and memory consumption at runtime. Their study revealed that Bison LALR (1) uses the least memory, while Knuth LR (1) uses the most. Knuth LR (1) runs the slowest and LR (0) the fastest.

Parser generation tools can be helpful apart from the compiler generation tasks. Renee McCauley [13] analyzed how these tools can be used in delivering different courses beyond compiler generation. Parser generation tools are an excellent example of principles that have stood out over time because the theory and methods that make them work are well understood and used to make different tools. McCauley claims that ANTLR is far more effective than the standard Lex/Yacc method because

it combines the lexer and parser, creates parse trees automatically, and has tree-walker grammars.

Several qualitative comparisons are made between the features supplied by Lex/Yacc and ANTLR. Still, research has yet to look at the empirical features, such as the productivity of language implementers and the tools' simplicity, intuitiveness, and maintainability. Ortin F. et al. [14] performed an empirical evaluation with Software Engineering majors at college. Two independent student groups implemented the same language with different parser generators and compared their performance across several features. According to the study, ANTLR has demonstrated significantly improved simplicity and maintainability.

With the rise of the World Wide Web, link context has become a common way to determine what a web page is about. Many ways have been tried to use the link context to figure out the exact context of the link, but they could have worked better. Gupta S. et al. [15] used the LALR parser to determine the link context. They collected different web pages and used a tag tree to find the concepts. They then used the Bison parser to figure out the link's context. Using the Jaccard coefficient, they compared their technique to one based on anchor text. Based on the results of the Jaccard coefficient, it is revealed that the Bison parser technique is better than other techniques.

Chien, L. R. et al. [16] presented DICE, an online automatic assessment system for test-based assignment tutoring and problem-solving. For plagiarism detection, the SableCC parser generator is used to build the abstract syntax tree (AST) from the student's answer and transform it into Polish Reverse Notation (PRN) for further assessment. After transforming the student's response into PRN form, they can be matched with the answers provided by other students and then assigned a grade. The DICE system was utilized in Hsing Kuo High School's programming course. The experiment revealed that it significantly reduces teachers' workload and encourages students to do their assignments independently.

László, Z. et al. [17] proposed MOFCOM, a development framework for reverse engineering. The parser built by the SableCC first parses the source code and generates an abstract syntax tree. During the semantic analysis stage, a name analysis is performed based on the syntax tree, which includes finding code declarations and releasing their references. The AST is then enriched with the necessary semantic information at the end of this stage. This AST is later utilized to generate the abstract model of the source code during the model creation stage. The model can be modified using the Model Transformer and turned back into source code using the Code Generator. The code generator was used successfully to generate formatted C# and XML files.

Chen, Andrew et al. [18] presented ACER, a framework for generating call graphs based on abstract syntax trees (ASTs). ACER employs the tree-sitter tool to establish compatibility with various programming languages. Our decision to prioritize generators that rely on abstract syntax trees (ASTs) is based on their efficiency and simplicity in specific situations. However, it is essential to note that a fully quantified intermediate representation generally offers more comprehensive

information while it involves compilation. To assess the efficacy of their system, the researchers developed two context-insensitive Java generators and conducted a comparative analysis with established open-source Java generators.

After conducting a comprehensive literature review on the parsers utilized for various software engineering tasks, it is evident that most studies focused on applying ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, and Bison to address domain-specific problems. Therefore, we have only selected these six parsers for further comparative analysis.

III. COMPARATIVE EVALUATION

The results of the parser comparison based on a comprehensive set of features are provided in this section. Section A defines the features selected for comparison. Section B compares the selected parsers based on these features.

A. Evaluation Parameters

When comparing different parsers, selecting a set of features based on which the parsers will be compared is essential. The following sections describe each selected feature in detail and explain why they should be considered when selecting a parser.

1) Parser Specifications:

The parser is the central component of every language processing tool. The result of the parsed source code relies on the parser's capabilities. Unparsed or incorrectly parsed parts of the source code will affect the parsed code. The following parser specifications must be taken into account when selecting a parser:

Parsing Algorithm and Parsing Strategy: The parsing algorithm and the parsing strategy refer to the specific technique and the overall approach the parser uses to analyze the input text and build a parse tree. The parsing algorithm is important in selecting a parser because it directly impacts its efficiency, error detection abilities, and support for handling different grammar structures.

Lookahead Tokens: Lookahead tokens refer to the number of tokens a parser can look ahead to in the input text to determine the appropriate action. The number and type of lookahead tokens directly impact the parser's ability to handle specific grammar and the level of ambiguity it can resolve.

Computational Complexity: Computational complexity refers to the time and memory required by a parser to analyze input text and build a parse tree. By evaluating the computational complexity, one can make informed decisions about whether the parser can handle larger inputs effectively, optimize performance, and meet the specific requirements of the parsing task.

Automatic Tree Generation: Automatic tree generation automatically constructs a structured representation of parsed input, such as a parse tree or abstract syntax tree, without manual intervention. It is essential as it enables further analysis, transformation, and interpretation of the parser data.

Left and Right Recursion: A production rule is said to be right-recursive if the non-terminal symbol appears at the end of the right-hand side of the production rule. In contrast, left recursion occurs when the leftmost symbol on the right-hand

side of a production rule is the same as the non-terminal symbol defined by that rule.

Incremental Parsing: Incremental Parsing involves parsing a document or a stream of data incrementally, as opposed to parsing it all at once.

Fault-Tolerant: Fault tolerant behavior allows the parser to continue functioning properly or degrade gracefully in the presence of faults, errors or unexpected behavior.

2) Grammar Specifications:

Grammar is the foundation for the parser to understand and process the input language effectively. The grammar specifications to be considered for parser selection are:

Input Grammar Notation: Input grammar notation refers to the formal representation of the grammar rules in a specific format, such as BNF or EBNF. The choice of notation can affect the grammar's readability, ease of use, and complexity.

Joint Lexer and Parser: Some parsers use one grammar notation for lexical and parsing purposes, while others use separate files for both tasks. Combining lexical analysis and parsing in the same file improves parsing speed and reduces memory usage by eliminating the overhead of intermediate file operations. Using separate files allows for modularity at the expense of increased I/O operations and resource utilization.

Lexer Type: Some parsers work with the generated lexer, while others use an external lexer. Choosing between a generated lexer optimized for the grammar and an external lexer can impact parser performance. A generated lexer improves tokenization and parsing efficiency, while an external lexer may introduce overhead through inter-component communication, potentially affecting overall performance.

Lexer Algorithm: The lexer algorithm refers to the lexer's procedure to match tokens in the input text. The lexer algorithm's choice can affect the tokenization process's speed and accuracy.

Semantic Predicates: Semantic predicates refer to the additional conditions that must be satisfied to apply a grammar rule. This feature can help the parser resolve ambiguities and produce more accurate parse trees.

Semantic Actions: Semantic actions refer to the additional code executed when a grammar rule is applied. This feature can help the parser generate output based on the input text, such as code or data structures.

3) Editor Specifications:

The primary role of an editor is not directly related to parsing. However, it can provide various features and functionalities supporting parsing activities such as syntax highlighting, auto-completion, error checking and debugging, code navigation, refactoring, and integration with parsing libraries and tools.

IDE Support: An IDE provides a comprehensive set of tools and features that aid in developing and debugging parsing activities. A robust IDE can greatly enhance developers' productivity working with a parser.

Plugin Support: Plugin support refers to the ability of the parser to integrate with third-party IDE plugins or extensions.

4) General Specifications:

Some general capabilities that should be considered while selecting a parser are:

License: The license of the parser should be compatible with the overall licensing requirements of the project. Some licenses impose restrictions on the distribution and redistribution of the software. A parser with a commercially friendly license should be selected for developing proprietary software.

Testing Tool: A testing tool assists in evaluating and validating the parser implementation's performance, functionality, and correctness.

B. Parser Evaluation

TABLE I compares ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, and Bison based on the selected features.

JavaCC generates LL(k) parsers, where LL stands for left-to-right. LL parsers are top-down parsers that parse the input from left to right, constructing a leftmost derivation of the input. The number k represents how many tokens the parser looks ahead to in the input stream to decide which production rule to apply next [19]. On the other hand, ANTLR uses the LL(*) parsing strategy, which is a more powerful variant of the LL(k) strategy. The LL(*) strategy used by ANTLR can handle grammars that are not LL(k) by performing an infinite lookahead to determine which grammar rule to apply [20].

Yacc and SableCC use the LALR(1) parsing technique to parse the input stream and generate a parse tree. LALR stands for Look-Ahead LR, a bottom-up parsing approach that uses a finite state to recognize the input tokens and a parse table to derive a parse tree [11]. LALR(1) means that the parser has a one-token look-ahead, which means it needs to read only one token ahead to decide which rule to apply in the parsing process.

Bison is a popular parser generator tool that can generate both LR(1) and LALR(1) parsers. However, Tree-Sitter only generates an LR(1) parser. An LR(1) parser is a bottom-up parser that can handle a larger grammar class than other bottom-up parsers. However, the LR(1) parser may have many states and transitions, which can increase the size of the parsing table [36]. Bison uses a modified LALR(1) algorithm that includes some LR(1) features, allowing it to handle a wide range of grammar while keeping the size of the parsing table relatively small [15].

The computational complexity of JavaCC is $O(n^k)$, where n is the length of the input stream, and k is the number of lookahead tokens, whereas ANTLR has a complexity of $O(n^4)$ [21]. The complexity of LALR and LR parsers like Yacc, Bison, Tree-Sitter, and SableCC is $O(n)$ [22]. Yacc and Bison use BNF notation to define their grammar [14], [23], while ANTLR, SableCC, and JavaCC use an extended version of BNF (Backus-Naur Form) [11], [20], [24]. Tree-Sitter does not use traditional grammar notations like BNF or EBNF. Instead, the grammar notation used by the Tree-sitter is based on a set of rules and patterns typically defined in JavaScript or JSON [36].

ANTLR, JavaCC, and SableCC use one grammar notation in a single file to define the lexical structure and the grammar rules for parsing the input language, which is suitable for small-to-medium-sized languages [25]-[26]. The lexer is generated automatically by them based on the lexer rules defined in the grammar file [14]. However, Tree-Sitter, Yacc and Bison use different grammar notation or parsing strategies for both parsing and lexical purposes suitable for large languages with many rules. For lexical analysis, Yacc and Bison use a separate tool called Lex (or Flex) to generate a lexer that recognizes individual tokens in the input language based on regular expressions specified in a separate file. Tree-Sitter generates lexer based on grammar rules and can even work with an external lexer [23], [27].

ANTLR, JavaCC, and SableCC support semantic predicates, whereas Tree-Sitter, Yacc and Bison don't [28]. Semantic actions in JavaCC and ANTLR can be embedded within the grammar rules or separated into code blocks [25]. However, in ANTLR, the semantic actions are typically embedded within the grammar rules rather than separating them from the grammar [29]. In SableCC and Tree-Sitter, semantic actions are separated from the grammar rules and are defined in a separate code block

or method [30]. In Yacc and Bison, semantic actions are embedded within the grammar [31].

LL parsers support right recursion but can't handle left recursive grammars. Therefore, JavaCC only supports right recursion [25]. However, ANTLR-4 overcomes this limitation and supports direct left recursion through grammar rewriting [21]. As the bottom-up parsers, Tree-Sitter, Yacc, Bison, and SableCC support both left and right recursion [32]-[33].

ANTLR and JavaCC produced clear error messages that include information about the specific error and the line where it occurs, which helps in troubleshooting and debugging [28]. Similarly, SableCC provides somewhat informative error messages. Yacc, Bison, and Tree-sitter can easily replace the default error messages with more informative ones to give users effective error feedback [37]. It's worth noting that Tree-sitter is the only parser that does not stop when an error occurs. However, other parsing tools may need to address this aspect to meet better the expectations of users who require fault tolerance in their parsing tasks. It also allows parsing a specific code section without re-producing the entire document. In contrast, this facility is not available in other parsers yet [18].

TABLE I. FEATURE COMPARISON OF LEADING PARSERS

Features	JavaCC	ANTLR	Tree-sitter	YACC	Bison	SableCC
Parsing Algorithm	LL(k)	Adaptive LL(*)	LR(1)	LALR(1)	LALR(1), LR(1)	LALR(1)
Lookahead Tokens	K	Adaptive Finite (*)	1	1	1	1
Computational Complexity	$O(n^k)$	$O(n^4)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Parsing Strategy	Top-down	Top-down	Bottom-up	Bottom-up	Bottom-up	Bottom-up
Automatic Tree Generation	Yes	Yes	Yes	Yes	Yes	Yes
Right Recursion	Yes	Yes	Yes	Yes	Yes	Yes
Left Recursion	No	Yes	Yes	Yes	Yes	Yes
Incremental Parsing	No	No	Yes	No	No	No
Fault-Tolerant	No	No	Yes	No	No	No
Input Grammar Notation	EBNF	EBNF	JavaScript DSL, JSON	BNF	BNF	EBNF
Semantic Actions	Embedded & Separate	Embedded & Separate	Separate	Embedded	Embedded	Separate
Semantic Predicates	Yes	Yes	No	No	No	Yes
Joint Lexer & Parser	Yes	Yes	No	No	No	No
Lexer	Generated	Generated	Generated + External	External	External	Generated
Lexer Algorithm	DFA	DFA	Custom	DFA	DFA	DFA
IDE Support	No	ANTLRWorks	No	No	No	No
Plugin Support	Eclipse, IntelliJ IDEA, NetBeans, Visual Studio Code	Eclipse, IntelliJ IDEA, NetBeans, Visual Studio Code	Neovim, Visual Studio Code	Eclipse, Vim, Visual Studio Code	Eclipse, Vim, Visual Studio Code	Eclipse, Vim, Visual Studio Code
Testing Tool	No	TestRig	No	No	No	No
License	Free, BSD	Free, BSD	Free, MIT	Free, BSD	Free, BSD, GNU GPL	Free, GNU GPL

ANTLRWorks is the official standalone IDE designed explicitly for ANTLR that provides a comprehensive set of features for developing and debugging parsers generated by ANTLR. There is no such IDE available for other parsers. However, several IDEs provide plugin support for various parsers. Eclipse, IntelliJ IDEA, NetBeans, and Visual Studio Code support ANTLR, JavaCC, and SableCC. Eclipse, Vim, and Visual Studio Code support Yacc and Bison, while NeoVim and Visual Studio Code provide plugin support to Tree-Sitter [29].

ANTLR provides a tool called "TestRig" that can be used to test the functionality of the generated parsers [14]. However, Tree-Sitter, Yacc, Bison, JavaCC, and SableCC need testing tools. ANTLR, JavaCC, and Yacc are distributed under the BSD-style license, SableCC is distributed under the GNU Lesser General Public License (LGPL), while Tree-Sitter is distributed under MIT. These licenses allow for free use, modification, and distribution of the software in commercial and non-commercial settings, as long as the copyright notice and disclaimer are included in the source code and documentation [29], [34]. However, Bison is licensed under BSD and GNU GPL [35].

IV. DISCUSSION, LIMITATIONS, AND FUTURE DIRECTIONS

Each parser has different features and limitations and is suitable for a particular situation based on its features. Therefore, developers frequently trade between various features when choosing a parser. Regarding parsing algorithms, LL(k) parsers like JavaCC are suitable for parsing simpler grammars with relatively few nonterminal and terminals with limited lookahead. LL(*) parsers like ANTLR are ideal for parsing more complex grammars with multiple viable paths. However, LL(*) parsers can be slower than LL(k). LALR(1) parsers like Yacc and SableCC can handle most context-free grammar reasonably efficiently but are more complicated to implement and may need help with complex grammar. Bison and Tree-sitter are suitable for more complex grammar that LALR(1) can't handle but requires more memory and can be slower. However, incremental parsing improves the overall parsing speed in tree-sitter.

Regarding computational complexity, the complexity of JavaCC increases exponentially with the number of lookahead tokens, limiting its scalability for the large k values. ANTLR's computational complexity is $O(n^4)$, meaning its running time increases faster than LL(k) parsers. However, ANTLR uses various optimization techniques, which reduce its complexity and improve performance. The computational complexity of Tree-sitter, Yacc, Bison, and SableCC increases linearly with the input size, making them highly efficient and scalable.

Regarding grammar specification, Yacc and Bison use BNF notation, which limits their expressiveness and makes grammar specification complex. Therefore, they aren't suitable for complex grammar. Whereas ANTLR, JavaCC, and SableCC use an extended version of BNF, which provides additional notation constructs, allowing for more expressive and explicit grammar definitions. Tree-sitter grammar is specified in JavaScript notation, allowing for more custom grammar specifications.

ANTLR, JavaCC, and SableCC support semantic predicates, so these parsers are suitable when parsed language syntax is

somewhat ambiguous and needs extra information to resolve ambiguity. However, Tree-sitter, Yacc and Bison parsers should be avoided because they don't support semantic predicates. For left recursive grammars, Tree-sitter, Yacc, Bison, and SableCC are preferred over top-down parsers like ANTLR and JavaCC as they support left recursive grammars. Whereas ANTLR and JavaCC are preferred over others for the right recursive grammar, as the right recursion in Tree-sitter, Yacc, Bison, and SableCC can increase stack usage and may slow down the parsing speed.

Although each parser has different features, ANTLR, JavaCC, and SableCC offer features that significantly impact the overall performance of the parser. These features include the parsing algorithm or the parsing strategy, extended grammar notation, and support for semantic predicates. Among these parsers, SableCC is somewhat challenging to understand and implement; however, it can handle a wide range of context-free grammars. Moreover, LALR parsers like SableCC can sometimes produce error messages that are not specific and user-friendly. This is because LALR parsers often identify errors based on the transition conflicts in the parsing table, which may not directly correspond to the source code constructs causing the issue. When an error is encountered, the parser may report the state, the expected tokens, or the conflict in the parsing table, which can be difficult for non-expert users to interpret.

ANTLR and JavaCC are easier to set up with little to no prior understanding of these parsers. They can provide error messages with more specific information about the nature and location of the error in the source code. In case of an error, an LL parser can often identify the exact token or production rule that caused the issue. Regarding grammar specification, JavaCC is helpful for simpler grammar as it can only look at a fixed number of tokens ahead to decide the following rule to apply. However, as ANTLR supports infinite tokens to look ahead, it allows complex grammar specification with more than one possible alternative path.

It can be argued that other parsers with enhanced features and capabilities might be available. However, our selection of parsers is based on their frequent utilization in the state-of-the-art literature. The widespread usage of these parsers across various problem domains indicates they have the most advanced parsing capabilities.

So far, our evaluation of these parsers has been limited to a theoretical examination of their features. However, in the future, we intend to extend this study by evaluating these parsers in a practical case study to see how well they perform.

V. CONCLUSION

This article compared six popular parsers (ANTLR, JavaCC, SableCC, Treesitter, Yacc, and Bison) to explore their features, strengths, and weaknesses in specific contexts. Since each parser offers unique features and capabilities, our research shows that the choice of parser must be made considering the software system's particular needs and the input data's characteristics. A parser's effectiveness, however, heavily depends on elements such as the parsing algorithm, the input grammar notation, and the support for semantic predicates. Popular parsing tools like ANTLR and JavaCC offer extended grammar definitions in

Extended Backus-Naur Form (EBNF) notation and support embedding semantic actions. The difference between JavaCC and ANTLR is that the latter permits the specification of sophisticated grammar with several alternative pathways. Researchers and practitioners in software engineering can benefit significantly from the comparative analysis offered in this article, which will help them choose the best parser for their needs.

ACKNOWLEDGMENT

This work is funded by the Higher Education Commission, Pakistan, through the NRPUR MRED project (20-15651).

REFERENCES

- [1] Zhou, J., Li, Z., & Zhao, H. (2019). Parsing all: Syntax and semantics, dependencies and spans. *arXiv preprint arXiv:1908.11522*.
- [2] M. Jain, N. Sehrawat, N. Munsri, "Compiler Basic Design And Construction," Proc. Journal of Computer Science and Information Technology, vol. 3, no. 10, Oct, pp. 850 – 852, 2014
- [3] Lu, Y. (2022). *MMP: An Object-Oriented Multi-Machine Parser Generator* (Master's thesis, ETH Zurich, Computer Engineering and Networks Laboratory, Distributed Computing Group).
- [4] NASSIRI, H., MACHKOUR, M., & HACHIMI, M. (2020). An Intermediate Representation-based Approach for Query Translation using a Syntax-Directed Method. *International Journal of Advanced Computer Science and Applications*, 11(8).
- [5] Kostalia, E. E., Petrakis, E. G., & Bourbakis, N. (2020, November). Evaluating methods for the parsing and understanding of mathematical formulas in technical documents. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)* (pp. 407-412). IEEE.
- [6] Baqai, H., Ilyas, A., & Ejaz, A. Comparison of Parsing Techniques.
- [7] Raju, C., Thirupathi, M., & Arvind, T. (2013). Analysis of Parsing Techniques and Survey on Compiler Applications. *International Journal of Computer Science and Mobile Computing (IJCSMC)*, 2(10), 115-125.
- [8] Straka, M., & Straková, J. (2017, August). Tokenizing, pos tagging, lemmatizing, and parsing ud 2.0 with udpipe. In *Proceedings of the CoNLL 2017 shared task: Multilingual parsing from raw text to universal dependencies* (pp. 88-99).
- [9] Cruz, D., Pereira, M. J., Berón, M., Fonseca, R., & Henriques, P. R. (2007). Comparing generators for language-based tools.
- [10] Pehlivan, H. (2019). Designing and interpreting a mathematical programming language. *Language*, 23(6), 1027-1041.
- [11] Anwar, M. W., Latifaj, M., & Ciccozzi, F. (2022, May). Blended Modeling Applied to the Portable Test and Stimulus Standard. In *ITNG 2022 19th International Conference on Information Technology-New Generations* (pp. 39-46). Cham: Springer International Publishing.
- [12] Chen, X., & Pager, D. (2011, May). Full LR (1) parser generator Hyacc and study the performance of LR (1) algorithms. In *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering* (pp. 83-92).
- [13] McCauley, R. (2001). A bounty of accessible language translation tools. *ACM SIGCSE Bulletin*, 33(2), 14-15.
- [14] Ortin, F., Quiroga, J., Rodríguez-Prieto, O., & Garcia, M. (2022). An empirical evaluation of Lex/Yacc and ANTLR parser generation tools. *Plos one*, 17(3), e0264326.
- [15] Gupta, S., & Yadav, S. (2014, February). A novel approach for link context extraction using Bison parser. In *2014 IEEE International Advance Computing Conference (IACC)* (pp. 941-945). IEEE.
- [16] Chien, L. R. DICE, a Parse Tree Based On-Line Assessment System.
- [17] László, Z., & Sulyán, T. (2006). Compiler Generator for Creating MOF-compliant Source Code Models. In *Software Engineering Research and Practice* (pp. 851-857).
- [18] Chen, Andrew, Yanfu Yan, and Denys Poshyvanyk. "ACER: An AST-based Call Graph Generator Framework." *arXiv preprint arXiv:2308.15669* (2023).
- [19] Ryu, S. (2016). Scalable framework for parsing: from Fortress to JavaScript. *Software: Practice and Experience*, 46(9), 1219-1238.
- [20] NASSIRI, H., MACHKOUR, M., & HACHIMI, M. (2020). An Intermediate Representation-based Approach for Query Translation using a Syntax-Directed Method. *International Journal of Advanced Computer Science and Applications*, 11(8).
- [21] Parr, T., Harwell, S., & Fisher, K. (2014). Adaptive LL (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10), 579-598.
- [22] Hester, J. R., & Shinan, E. (2021). Lerche: Generating data file processors in Julia from EBNF grammars. *Journal of Open Source Software*, 6(64), 3497.
- [23] Zhang, Y., Lu, Y., & Yang, B. (2017, June). Parsing statement list program using flex and bison. In *2017 First International Conference on Electronics Instrumentation & Information Systems (EIIS)* (pp. 1-4). IEEE.
- [24] Ryu, S. (2016). Scalable framework for parsing: from Fortress to JavaScript. *Software: Practice and Experience*, 46(9), 1219-1238.
- [25] Maniscalco, M. (2006). Paradigms for using semantic actions and code execution with JavaCC.
- [26] Cerveille, J., Forax, R., & Roussel, G. (2006, August). Tatoo: an innovative parser generator. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java* (pp. 13-20).
- [27] Mahmood, S. A., & Melhum, A. I. (2012). DESIGN ASSEMBLER BASED ON LEX AND YACC. *Iraqi Journal of Science*, 53(1).
- [28] Parr, T., & Fisher, K. (2011). LL (*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices*, 46(6), 425-436.
- [29] Bovet, J., & Parr, T. (2008). ANTLRWorks: an ANTLR grammar development environment. *Software: Practice and Experience*, 38(12), 1305-1332.
- [30] Cerveille, J., Forax, R., & Roussel, G. (2006, August). Tatoo: an innovative parser generator.
- [31] Wu, X. (2007). *Component-based language implementation with object-oriented syntax and aspect-oriented semantics*. The University of Alabama at Birmingham.
- [32] Younessi, O., Bahrololoomi, M. H., & Yazdani, A. (2013, May). The extension of deterministic cancellation parser to directly handle indirect and hidden left recursion. In *2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)* (pp. 1-4). IEEE.
- [33] Scott, E., & Johnstone, A. (2005). Generalized bottom up parsers with reduced stack activity. *The Computer Journal*, 48(5), 565-587.
- [34] Sgaglione, L., Papale, G., Mazzeo, G., Cerullo, G., Starace, P., & Campanile, F. (2016, April). Data Collection Framework. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science-Volume 1 and 2* (pp. 374-379).
- [35] German, D. M., Di Penta, M., & Davies, J. (2010, June). Understanding and auditing the licensing of open source software distributions. In *2010 IEEE 18th International Conference on Program Comprehension* (pp. 84-93). IEEE.
- [36] Beckmann, Tom, et al. "Partial Parsing for Structured Editors." Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, 2022.
- [37] Jeffery, Clinton L. "Generating LR syntax error messages from examples." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25.5 (2003): 631-640.