Evaluation of the Use of Different Parser Generators in a Compiler Construction Course

Francisco Ortin[1,2](✉), Jose Quiroga[1], Oscar Rodriguez-Prieto[1], and Miguel Garcia[1]

[1] Computer Science Department, University of Oviedo, C/Federico Garcia Lorca 18, 33007 Oviedo, Spain
ortin@uniovi.es

[2] Cork Institute of Technology, Computer Science Department, Munster Technological University, Rossa Avenue, Bishopstown, Cork T12 P928, Ireland

Abstract. Compiler construction is a common course in many computer science degrees. Although there are different lexer and parser generation tools, Lex/Yacc is still the preferred option in multiple universities. ANTLR is a powerful compiler construction tool that, among other features, provides parser generation. Although ANTLR has been used in the implementation of different languages and software tools, it is not as common as Lex/Yacc in compiler construction courses. Therefore, we conduct an experiment to evaluate the impact of using ANTLR instead of Lex/Yacc in courses where students build a compiler. To that aim, we divide the year-3 students of a Programming Language Design and Implementation course into two random groups. The course is delivered using Lex/Yacc for one group and ANTLR for the other one. Different values are measured to statistically compare both approaches, including the work completed by the students, the number of hours it took them to finish different tasks, their anonymous opinion about the parser generator used, and their performance. The use of ANTLR shows significant benefits in most of the evaluations.

Keywords: Parser generation · Compiler construction · Parser · Lexer · ANTLR · Lex · Yacc

1 Introduction

Compiler construction courses are widespread in computer science degrees. When building a compiler, it is common to use parser and lexer generation tools instead of creating them manually [1]. With these tools, the language implementor describes the lexical/syntax specification with grammars, and the lexer/parser generators produce the implementation of the language specified [2].

There exist plenty of parser generators, such as Yacc/Bison, ANTLR, JavaCC, LISA, SableCC, and Coco/R, among many others [3]. Although these tools provide more advanced features than the classic Lex/Yacc approach, Lex/Yacc seems to be the preferred option when delivering compiler construction courses. Table 1 shows the parser generation tools used in the compiler construction courses of the top 10 universities according to Shanghai Consultancy [4], Quacquarelli Symonds [5], and Times Higher Education [6] university rankings. We can see how most courses (12 out of 14) use variants of the classic Lex/Yacc approach (Flex/Bison, OCamlLex/OCamlYacc, Menhir, ML-Lex/ML-Yacc, and JFlex/CUP are different versions of Lex/Yacc).

Table 1. Parser generation tools used in the compiler construction courses of top computer science universities (alphabetically ordered), according to Shanghai Consultancy, Quacquarelli Symonds, and Times Higher Education rankings.

University	Course name	Parser generator
Carnegie Mellon University	Compiler design	ML-Lex/ML-Yacc
ETH Zurich	Compiler design	OCamlLex/Menhir
Harvard University	Compilers	OCamlLex/Menhir
Imperial College London	Compilers	Lex/Yacc
Massachusetts Institute of Technology	Computer language engineering	ANTLR
Nanyang Technological University	Compiler technology of programming languages	Lex/Yacc
National University of Singapore	Compiler design	JFlex/CUP
Stanford University	Compilers	Flex/Bison
Swiss Federal Institute of Technology in Lausanne	Computer language processing	Manual implementation with a LL(1) grammar
Tsinghua University	Principles of assembly and compilation	Lex/Yacc
University of California, Berkeley	Programming languages and compilers	JFlex/CUP
University of Cambridge	Compiler construction	OCamlLex/OCamlYacc
University of Oxford	Compilers	OCamlLex/OCamlYacc
University of Toronto	Compiler and interpreters	Flex/Bison

The only course that uses a parser generator distinct to Lex/Yacc is the Computer Language Engineering course delivered at MIT. They use ANTLR, a powerful top-down parser generator used in the development of different software products such as Hibernate, Twitter search, Hadoop, and Oracle SQL developer IDE [7].

ANTLR follows a top-down parsing strategy, and implements an adaptive LL(*) parsing algorithm [8] that analyzes a finite but not fixed number of tokens (lookahead) [9]. At parsing, it launches different subparsers when multiple productions could be derived, exploring all the possible paths. Then, the subparser with the minimum lookahead depth that uniquely predicts a production is selected [9].

On the contrary, Yacc is a bottom-up parsing tool that implements an LALR(1) (Look-Ahead LR) parser, a simplification of general LR(1) bottom-up parsers [10]. To know whether the parser must analyze another token from the input or reduce one production from the grammar, the algorithm analyzes one single token (lookahead).

Yacc uses the BNF (Backus-Naur Form) notation to specify language grammars [11], while ANTLR supports an extended version that includes different operators including those provided in common regular expressions [7]. This feature allows language implementors to describe grammars with fewer productions. Moreover, simpler grammars produce smaller parse trees, which are automatically generated by ANTLR (Yacc does not support this feature). ANTLR uses the same grammar notation and parsing strategy for both lexical and syntactic language specifications. Yacc only provides syntax specification, while lexical analysis should be implemented apart, commonly using Lex, which generates Deterministic Finite Automata (DFA) [10].

ANTLR provides listeners to decouple grammars from application code, encapsulating its behavior in one module apart from the grammar, instead of fracturing and dispersing it across the grammar. Unfortunately, Yacc does not provide this option, so the user must write the semantic actions embedded across the grammar. Both tools provide right recursion. ANTLR 4 supports direct left recursion but not indirect one (Yacc does).

Since ANTLR provides more advanced features than Lex/Yacc [12], we conduct an experiment to analyze the impact of changing the lexer/parser generation tool in a compiler construction course (the main contribution of this article). We divide into two random groups the students of a Programming Language Design and Implementation course in a Software Engineering degree. Each group is taught a different parser generator and requested to develop an imperative procedural programming language. We then measure different variables such as the students' performance when using each tool and their opinion about the parser generator used.

The rest of the article is structured as follows. Section 2 presents the methodology used in our evaluation, and Sect. 3 details the results. Section 4 discusses the related work, and conclusions and future work are presented in Sect. 5.

2 Methodology

Our experiment was conducted with year-3 undergraduate students of a Programming Language Design and Implementation course [13] in a Software Engineering degree [14] at the University of Oviedo. The 6 ECTS course is delivered along the second semester, through 15 lectures and 14 for laboratory classes (two hours for each lecture and lab). Lectures introduce the theoretical concepts of programming language design. Labs follow a project-based learning approach, and they are mainly aimed at implementing a compiler of an imperative programming language. The language must provide integer, real and character built-in types, arrays, structs, functions, global and local variables, arithmetical, logical and comparison expressions, literals, identifiers, and conditional and iterative control flow statements [13]. The compiler is implemented in the Java programming language.

The course has two parts. In the first seven weeks, the students have to implement the lexical and syntactic analyzers of the mentioned programming language, using a generation tool. At the end of the first part of the course, their parser should return an Abstract Syntax Tree (AST) if no errors are detected for the input program. In the second part of the course, they traverse the AST to perform semantic analysis and code generation for a stack-based virtual machine [15].

The evaluation consists of a theory exam and a lab exam where the students must extend their language implementation. Both evaluations have the same weight and they are performed after delivering all the lectures and labs.

Our study took place in the second semester of the 2020–2021 academic year. The 183 students (131 males and 52 females) enrolled in the course were divided into two random groups: the first group had 92 students (67 males and 25 females) and the second one was 91 (64 males and 27 females).

For the first part of the course (the first seven weeks), lectures and labs for the first group of students were delivered using BYaccJ/JFlex (Java versions of the classic Yacc/Lex generators). For the second group, ANTLR was used instead. The second part of the course has the same contents for both groups, since there is no dependency on any generation tool. Both groups implemented the very same programming language, and they had the same theory and lab exams.

We first analyze the completion level of students' work after each lab related to the use of lexer and parser generators. We ask them to write down the number of additional autonomous hours it took them to finish their work, if they do not manage to complete it in the two hours lab. After each lab, we asked them to fill in an anonymous questionnaire about their opinion regarding the lexer/parser generator used, with the following questions:

- Q1: I have found it easy to use the tool(s) to generate the lexical and syntactic analyzers.
- Q2: The tool(s) used to generate the lexical and syntactic analyzers are intuitive.
- Q3: Eventual modifications of my language specification (lexical and syntactic) will be easy to perform due to the lexer and parser generation tool(s) used.

Besides measuring the students' performance and opinion after the labs, we also consider their final performance. To this aim, we compare the pass and fail rates of both groups (Lex/Yacc and ANTLR) and the number of students taking the lab exam. We also measure and compare the final marks of the students. We want to see if there are statistically significant differences between the values of the two groups of students. Since the two groups are independent and students' marks are normally distributed (p-value > 0.1 for all the distributions), we apply an unpaired two-tailed t-test ($\alpha = 0.05$).

3 Results

Figure 1 shows the percentage of work that the students manage to complete in the lab, and the number of additional hours it took them to finish their work. For all the labs analyzed (lexer and parser implementation and AST construction), more ANTLR students were able to finish their work in the lab. The ANTLR group presents from 4.5% to 18.3% higher completion percentages than the students of the Lex/Yacc group.

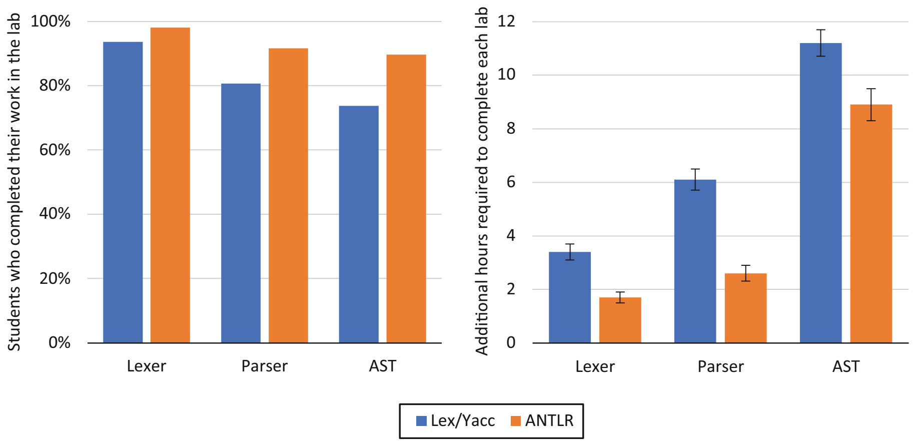

Fig. 1. Left-hand side: percentage of students that completed their work in the lexer, parser and AST creation labs (2 h each). Right-hand side: number of additional hours required to complete each lab (average valuers are shown; whiskers represent 95% confidence intervals).

The right-hand side of Fig. 1 shows the number of additional hours it took the students to complete each lab autonomously. Differences between both groups are statistically different (i.e., p-values for the unpaired two-tailed t-tests are lower than 0.01 and 95% confidence intervals do not overlap [16]). Lex/Yacc students needed, on average, from 1.7 to 3.5 more hours than the ANTLR group.

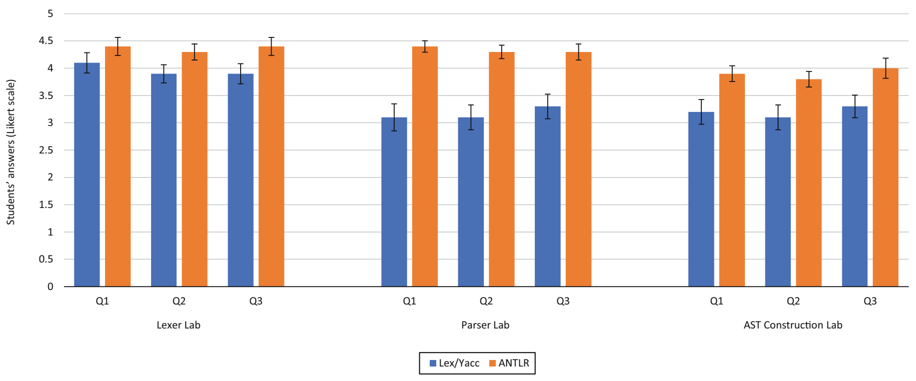

Fig. 2. Average answer values for the three questions asked. Answers are on a 5-point Likert scale, ranging from 1 = "completely disagree" to 5 = "completely agree". Whiskers represent 95% confidence intervals.

In Fig. 2, we summarize the answers to the questionnaire filled in by the students. The arithmetic mean of the ANTLR group is higher than Lex/Yacc, for all the questions and labs. After using different lexer and parser generators, the students think that the simplicity (Q1), intuitiveness (Q2), and maintainability (Q3) capabilities of ANTLR are greater than those for Lex/Yacc. The greatest differences appear when they implement a syntax analyzer with a parser generator, whereas lexer implementation is the lab with the smallest differences. All the differences but the simplicity of the lexer generation (Q1) after the lexer construction lab are significantly different (95% confidence intervals do not overlap).

Figure 3 presents exam attendance, pass and fail rates, and the final marks for the two groups of students. Those students who used ANTLR show 11.6% higher pass rate than the Lex/Yacc group. Likewise, ANTLR students attended the exam 16.5% more than the students who used Lex/Yacc. This is something important, because the Programming Language Design and Implementation course has quite low rates of students taking the exams. Its average value for the past courses is 66.5%, while the ANTLR group reached 89.9%. Differences between Lex/Yacc and ANTLR pass and fail rates, and exam attendance are significantly different (p-value < 0.01).

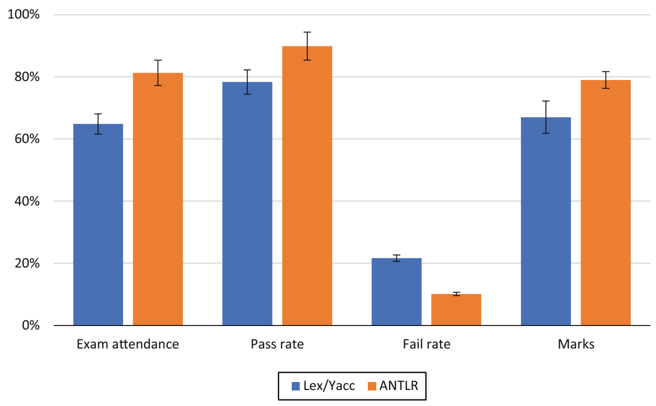

Fig. 3. Exam attendance, pass rate, fail rate and final marks of the Lex/Yacc and ANTLR groups of students.

Comparing the students' final marks, we can see in Fig. 3 that the students of the ANTLR group obtained significantly higher marks than those in the Lex/Yacc group. On average, the ANTLR students obtained 12% higher marks than Lex/Yacc. This leads us to think that ANTLR eases the implementation of the programming language. It might also help understand the concepts of language implementation, but most students who fail is because they do not pass the lab exam.

4 Related Work

The study undertaken by McCauley presents a shallow analysis of parser generation tools and how they could be used for different courses beyond compiler construction, such as operating systems, software engineering, and even human-computer interaction [12]. Parser generation tools represent a good example of principles that have endured, because the underlying theory and algorithms are well understood and used to implement different kinds of tools. McCauley indicates that ANTLR is more powerful than the traditional

Lex/Yacc approach, mainly because of its lexer and parser integration, automatic parse tree generation, and its tree walker grammars. No empirical evaluation is presented.

The work in [17] performs a qualitative and quantitative comparison of lexer (Lex, Flex, and Rex) and parser generators (Yacc, Bison, PGS, Lalr, and Ell). First, a qualitative comparison of both kinds of tools is presented by analyzing their features. Then, the sizes of the lexer/parser table and the generated program are compared, together with generation time, and execution time of the code produced. For lexical analysis, Rex is the tool with the lowest generation times and table and lexer sizes. It also generates the fastest lexers. For the parser generation tools, Bison provides the lowest generation times, and table and parser sizes, whereas Ell generates the fastest parsers.

Chen implements a Yacc-compatible open-source parser generator, called Hyacc, that accepts all LR(1) grammars and generates full LR(0), LALR(1), LR(1), and partial LR(k) parsers [18]. With Hyacc, an empirical comparison of runtime performance and memory consumption of the most common LR(1) parsing algorithms is presented [19]. 17 simple grammars and 13 specifications of real programming languages are used in the evaluation. The Knuth LR(1) algorithm uses more memory than the rest of the algorithms, and Bison LALR(1) is the one with the lowest memory consumption. Regarding runtime performance, Knuth LR(1) is the slowest algorithm and LR(0) runs the fastest.

An empirical comparison of Yacc, LISA, and ANTLR 3 parser generators is presented in [20]. The authors implement the Lavanda Domain Specific Language (DSL) as an interpreter with each of the three tools. The Lavanda semantics is specified as an attribute grammar that is translated to Yacc, LISA, and ANTLR 3. Based on the three implementations, a qualitative evaluation is presented where they analyze different features divided into three groups: language specification, parser generator, and the code produced. Overall, they state that Yacc is the poorest tool, and ANTLR 3 and LISA are similar, according to the qualitative comparison criteria utilized in their study. No evaluation of the time spent in all the implementations is presented.

Language workbenches are tools that lower the development costs of implementing new languages and their associated tools [21]. They include different components, such as parser generators, to make language-oriented programming environments practical [22]. An empirical evaluation measures the implementation of a tiny Questionnaire Language (QL) to create questionnaires forms using different language workbenches [23]. First, they analyze to what extent each language workbench is able to support all the features of the QL language. Then, they measure the source lines of code and the number of model elements used.

In [24], six experts are selected for an empirical analysis of tools for DSL implementation. The experts implement WAEBRIC, a little language for XHTML markup generation used for website creation. Each expert develops a different language implementation: Java, C#, and JavaScript implementation with no tools; and the other three using ANTLR, Microsoft "M", and OMeta. For the comparison, they measure different quantitative variables such as volume (number of modules, units, and lines of code), cyclomatic complexity, and percentage of code duplication. They also present a qualitative evaluation comparing the maintainability of each implementation.

5 Conclusions

Our evaluation of the impact of using ANTLR has shown that this tool provides significant benefits for the implementation of a medium-complexity imperative programming language, compared to Lex/Yacc. We have statistically measured significant differences in the time spent to implement lexers, parsers, and AST construction. The students state that ANTLR is simpler, more intuitive, and language specifications are more maintainable. Those students who used ANTLR exhibited not only higher pass rates but also took more exams. Their average final marks were significantly higher than the students who used Lex/Yacc.

Our study could be extended with other parser generators [3] and more sophisticated tools for compiler construction such as tree walkers [7], type checker generators [25, 26], and language implementation frameworks [27]. It could also be studied the impact of these tools to implement different kinds of programming languages, data formats, and implementation languages.

Acknowledgments. This work has been partially funded by the Spanish Department of Science, Innovation, and Universities: project RTI2018-099235-B-I00. The authors have also received funds from the University of Oviedo through its support to official research groups (GR-2011-0040).

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley, Boston (2006)
2. Parr, T.J., Quong, R.W.: ANTLR: a predicated-LL(k) parser generator. Softw.: Pract. Exp. **25**(7), 789–810 (1995)
3. Comparison of parser generators. https://en.wikipedia.org/wiki/Comparison_of_parser_generators. Accessed 19 Oct 2021
4. Shanghai Consultancy, Global Ranking of Academic Subjects, Computer Science. https://www.shanghairanking.com/rankings/gras/2021/RS0210. Accessed 19 Oct 2021
5. Quacquarelli Symonds, QS World University Rankings - Computer Science. https://content.qs.com/wur/Computer_Science.htm. Accessed 19 Oct 2021
6. Times Higher Education, World University Rankings 2021 by subject: Computer Science. https://www.timeshighereducation.com/world-university-rankings/2021/subject-ranking/computer-science#!/page/0/length/25/sort_by/rank/sort_order/asc/cols/stats. Accessed 19 Oct 2021
7. Parr, T.: The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf, Raleigh (2013)
8. Parr, T., Fisher, K.: LL(*): the foundation of the ANTLR parser generator. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 425–436. Association for Computing Machinery, New York (2011)
9. Parr, T., Harwell, S., Fisher, K.: Adaptive LL(*) parsing: the power of dynamic analysis. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, pp. 579–598. Association for Computing Machinery, New York (2014)
10. DeRemer, F., Pennello, T.: Efficient computation of LALR(1) look-ahead sets. ACM Trans. Program. Lang. Syst. **4**(4), 615–649 (1982)

11. Levine, J.R., Mason, T., Brown, D.: Lex & Yacc, 2nd edn. O'Reilly & Associates Inc., USA (1992)
12. McCauley, R.: A bounty of accessible language translation tools. ACM SIGCSE Bull. **33**(2), 14–15 (2001)
13. Ortin, F., Zapico, D., Cueva, J.M.: Design patterns for teaching type checking in a compiler construction course. IEEE Trans. Educ. **50**, 273–283 (2007)
14. Ortin, F., Redondo, J.M., Quiroga, J.: Design and evaluation of an alternative programming paradigms course. Telemat. Inform. **34**(6), 813–823 (2017)
15. Ortin, F., Redondo, J.M., Perez-Schofield, J.B.G.: Efficient virtual machine support of runtime structural reflection. Sci. Comput. Program. **74**(10), 836–860 (2009)
16. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA, pp. 57–76. ACM, New York (2007)
17. Grosch, J.: Generators for high-speed front-ends. In: Hammer, D. (ed.) CCHSC 1988. LNCS, vol. 371, pp. 81–92. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51364-7_6
18. Chen, X.: Hyacc parser generator. http://hyacc.sourceforge.net. Accessed 19 Oct 2021
19. Chen, X., Pager, D.: Full LR(1) parser generator Hyacc and study on the performance of LR(1) algorithms. In: Proceedings of the 4th International C* Conference on Computer Science and Software Engineering, C3S2E 2011, pp. 83–92. Association for Computing Machinery, New York (2011)
20. da Cruz, D., Varanda, M.J., Verón, M., Fonseca, R., Henriques, P.R.: Comparing generators for language-based tools. Technical report, Departamento de Informatica, Universidade do Minho, CCTC, Braga, Portugal (2007)
21. Fowler, M.: Language workbenches: the killer-app for domain specific languages?. http://martinfowler.com/articles/languageWorkbench.html. Accessed 19 Oct 2021
22. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. https://resources.jetbrains.com/storage/products/mps/docs/Language_Oriented_Programming.pdf. Accessed 19 Oct 2021
23. Erdweg, S., et al.: Evaluating and comparing language workbenches: existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015)
24. Klint, P., van der Storm, T., Vinju, J.: On the impact of DSL tools on the maintainability of language implementations. In: Proceedings of the 10th Workshop on Language Descriptions, Tools and Applications, LDTA 2010, pp. 1–9. Association for Computing Machinery, New York (2010)
25. Ortin, F., Zapico, D., Quiroga, J., Garcia, M.: TyS - a framework to facilitate the implementation of object-oriented type checkers. In: Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, pp. 150–155 (2014)
26. Ortin, F., Zapico, D., Quiroga, J., Garcia, M.: Automatic generation of object-oriented type checkers. Lect. Notes Softw. Eng. **2**(4), 288–293 (2014)
27. Wimmer, C., Würthinger, T.: Truffle: a self-optimizing runtime system. In: Leavens, G.T. (ed.) Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH 2012, pp. 13–14. ACM, Tucson (21–25 October 2012)