Bulletin of the *Transilvania* University of Braşov
Series III: Mathematics and Computer Science, Vol. 3(65), No. 2 - 2023, 159-168
https://doi.org/10.31926/but.mif.2023.3.65.2.14

CONSIDERATIONS ON EFFICIENT LEXICAL ANALYSIS IN THE CONTEXT OF COMPILER DESIGN

Alexandra BĂICOIANU[*,1] **and Ioana PLAJER**[2]

Dedicated to Professor Radu Păltănea on the occasion of his 70th anniversary

Abstract

Each programming language needs a mechanism of translating source code into code understood by the computer, i.e., machine code or assembly. This is done by a compiler. A compiler is thus a fundamental tool for software development. Lexical analysis, as the first step of compilation, has an important role and should be performed in an optimal way. But not only compilers rely on lexical analysis. Its role lies nowadays also in pattern recognition or natural language processing.

In this paper, we present some of the main aspects of lexical analysis and how to efficiently perform it. Our aim is to offer some guidelines in constructing a modern and efficient scanner (lexer), outlining the criteria, which should be considered, presenting some pitfalls, and comparing existent tools for compiler construction. The main contribution of the paper is to offer practical guidelines and recommendations, especially to graduate students and programmers, in constructing an efficient scanner (lexer), considering also the proper tools currently available.

2000 *Mathematics Subject Classification:* 68Q01, 68Q45, 03D05, 68N20.

Key words: scanner, lexical analysis, deterministic finite automata (DFA), modern instruments, compilers and interpreters.

1 Introduction

Computers and humans are similar in the sense they both acquire and process information and at the same time they use that information and the deductions they can draw from it to handle everyday tasks. As lexical analysis is the first phase of the compilation process, it handles the processing of input language and it may be only one part of the compiler process, but it is the foundation of the

[1*] *Corresponding author*, Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: a.baicoianu@unitbv.ro

[2]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: ioana.plajer@unitbv.ro

entire process. According to cognitive research, word recognition is the first step in how humans perceive language. Understanding any language cannot happen at all without this stage.

Several examples from cognitive science, linguistics, and natural language processing area reveal that lexical choice can affect audience insight [1], [2], [3]. so it makes perfect sense to study the impact that a good lexical analysis can have. Proper analysis of the text allows computers to extract usefully structured information and pinpoint conclusions faster. There is an ongoing concern related to the compilation stages and everything that involves the optimal design of a compiler, as presented in studies namely [4], [5], and [6] where some criteria to consider are highlighted.

Despite all of the previous work, we still do not have any universal techniques for finding the best answer in terms of lexical examination. Given the fact that there are other lexer and parser generating tools - the classic Lex/Yacc based on C/C++, or the Java oriented JFLex [7], ANTLR is a robust compiler creation tool that, among other things, offers parser generation [8], [9].

In this study we proposed various criteria to be taken into account when performing a lexical analysis, and in addition we compared various specific tools for lexical analysis, precisely to propose an optimal variant of tool for this purpose.

This current work analyzes various problems in compiler design and implementation. The research, which was written for professionals and graduate students, focuses on a broad range of potential code optimizations, establishing the relative significance of a reliable scanner, and choosing the most efficient implementation strategies. The main contribution is the comparison of different existing tools and strategies, offering specific recommendation for an appropriate choice.

2 Theoretical aspects and background

The most frequent use of lexical analysis is in the context of compiler construction. As such, the theoretical background will be focused on this issue, with the mention that, in most cases lexical analysis for other tasks will be performed in a similar way, but taking into account the specific goal.

As the actor behind the scene, the compiler - a translator between the source code, in whichever programming language, and the computer - has an important contribution to the overall efficiency and usability of a programming language and therefore a significant impact on a good final product.

The tasks of a compiler are: take the source code, analyze the character stream and build tokens (lexical analysis), from the tokens, based on some grammars, build the parse tree (syntactic analysis) and finally generate the executable code in the base language understood by the computer.

The first module of a compiler is thus the lexical analyzer, or the scanner. The main stages of a scanner are: read input from file (section 2.1), group the characters from the input into tokens, which will be passed to the parser (section 2.2), report lexical errors (section 2.3) and build the symbol table (section 2.4).

While sections 2.1 and 2.2 are needed in any lexical analysis task, sections 2.3 and 2.4 are mainly specific for compiler construction.

There are two ways to build a scanner / compiler:

1. Handcrafted compiler, in which the programmer does all the work up to scratch;

2. Compiler generator: LEX and Bison, ANTLR, JFlex and JavaCC or similar tools.

2.1 Read the input

In order to implement a handcrafted scanner, the programmer has to first decide how to read the input from the file and how to separate the input string into substrings (lexemes), representing the tokens.

Languages such as C++ or Java offer some tools to separate a string into substrings, like StringTokenizer. These tools are not usable in the context of lexical analysis, as they need separators, for segmenting the text. These separators will not be part of the substrings. But in a program, all the separators which could be used in StringTokenizer, represent for the most part tokens for themselves. Furthermore, there are some tokens, like strings, which will falsely be separated into substrings by such a tool.

By consequence when performing lexical analysis, it is necessary to scan the input string sequentially, character by character and fit substrings on given patterns. This can be done using a buffer of fixed dimension N, divided into halves. Each time the lexer reaches the end of one half, it updates the other one, so that no lexeme is broken apart. This means that no lexeme should be longer than $\frac{N}{2}$. If the programming language does not permit a lexeme to break from one line to another then the input could be read line by line, by some readline instruction, thus avoiding the double buffering. Also, some lexers read the entire source file in the RAM memory, but that could be inefficient for a large source code.

2.2 Group the characters from the input into tokens

The characters of the input string are grouped into lexical units, called tokens. These lexical units have to be defined in a suitable way according to the specific language and then described by a pattern. According to these patterns each substring (lexeme) in the source code representing the token can be identified. The most common way of describing these patterns are regular expressions. Matching a lexeme on a pattern can be done in different ways:

(a) Detecting the tokens using the starting character of the substring is very simple to implement but not very easy to maintain. Small changes in the designed language for which the parser is built might result in big changes in all the lexer. Another disadvantage is that different tokens could eventually start with the same character, so the program would have to do more than one check for the same string sequence by reading ahead a number of characters and backtracking in case of failure. This would result in a loss of efficiency. For example, if there are two different tokens for integer and floating-point numbers, in case of a number like 4520.65, the first four characters apply to both tokens, so only when finding the character '.' the scanner can differentiate between integer and floating point. It has to look ahead four characters.

(b) Identifying tokens by deterministic finite automata (DFA) is one of the most used techniques and requires the construction of transition diagrams (TD) or DFAs for all the tokens. There are some good descriptions of how to build and implement such diagrams / automata in [10] and [11]. In the second one, there are also some indications of optimizing the transition table for the DFA. Using DFAs makes the maintenance of a scanner easier.

Generally, there are two ways to implement the DFAs:

- Table driven scanners which encode the DFAs transitions in one or more tables. Usually, lexer generators create such scanners;
- Direct-coded scanners which explicitly code the DFAs rules. This is often the approach of hand coded scanners.

(c) Detecting tokens with tools like regex may seem to the inexperienced programmer a simple and efficient way to construct a scanner. The regex library enables fitting some strings on given regular expressions. Thus, by designing appropriate regular expressions for each token it should be easy to use regex for the tokenization of a string representing the source code.

This approach is not as efficient as expected. The problem is, that regex engines do not use DFAs, working rather in a backtracking manner [12], [13]. This would result in an unwanted large complexity of the task, making it unusable for real compilers. It also can be difficult to correctly find the tokens, if more than one pattern matches one of the regular expressions. As a conclusion, this way of token detection should not be taken into consideration.

2.3 Report lexical errors

There are relatively few lexical errors that can be reported. Most of the errors occur at parsing time. Nevertheless, there can be some unrecognized characters, or some character strings, which do not match any pattern defined for the tokens. For a good error report, the scanner should be able to keep track of the line and column in the source code.

2.4 Build the symbol table

The symbol table generally stores the identifiers, with some necessary information. Sometimes it also can store the keywords, making it easier to separate

Quex	ANTLR	Lex	JFlex/JLex
C++ Python, C#	Java, Java script	C	C, C++, Java

Table 1: Generated code by each of the tools

Quex	ANTLR	Lex	JFlex/JLex
- easy to define - enables many set operations on regular expressions (similar to sets) - high flexibility in pattern definition	- easy to define	- easy to define - enables union and difference for character classes - less flexible compared to Quex and ANTLR	- easy to define - really similar with Lex

Table 2: Comparison on regular expressions

them from identifiers. A symbol table should provide fast access and is usually implemented by hashing [10].

The early compilers were all handcrafted. From the above stated problems, it should be obvious that constructing an efficient compiler by hand is not a trivial task. As the need for quick development, intermediate testing and other usage of lexical analysis grows, a series of tools to automate this task were created.

3 Methods: an exploration on lexical analysis

3.1 Comparison of available tools

We consider that a comparison of the most common compiler construction tools is very important and beneficial. In the context of this study, we established some comparison criteria to observe the differences between the most well-known tools used for lexical analysis, specifically Quex, ANTLR, Lex and JFlex/JLex.

In terms of the generated code, each of these tools generate code in C, C++, Java, Java Script or even Python, which implements the user's lexical analyzer. Table 1 contains a list of most frequent available programming languages.

Lexers are specified by regular expressions, so this criterion is absolutely obvious for our comparison. Regular expressions provide a simple way to describe finite regular languages by patterns for finite strings of symbols. From the point of view of regular expressions, the comparison is illustrated in table 2.

Generally, a lexer is combined with a parser, which together analyze the syntax of the source code of the program. But there are situations when these components are in separate files (and communicate with each other), or they can be combined into a single file. From this point of view, Quex is using separate files where the parser calls the lexer, but for ANTLR it can be written in the same file/grammar

Quex	ANTLR	Lex	JFlex/JLex
- enables multiple lexical modes and enables inheritance between modes	- enables multiple lexical modes	- does not enable multiple lexical modes	- does not enable multiple lexical modes

Table 3: Comparison on lexical modes

(easy to use together). Similar to Quex, Lex and JFlex/JLex are working with separate files.

The generation and manipulation of tokens is another criterion that should be considered in terms of performance and quality. Therefore, for the considered tools, we share some remarks related to tokens. For Quex, the tokens can be automatically / manually defined, further allowing the definition of a customized token class. Tokens for ANTLR can be defined automatically in the parser from characters if the lexer and the parser are in the same file, and in addition can be manually defined. ANTLR also allows the definition of a customized token class. Users can define tokens by using regular expressions in Lex. Concerning JFlex/JLex, the programmer specifies the tokens to match using regular expression rules. Furthermore, the scanning method can be customized, including by redefinition of the name and return type of this method. It is possible to specify exceptions that might be thrown during one of the specification's actions. The scanning method will be declared as returning values of class Yytoken if no return type is supplied.

The concept of lexical analyzer modes allows to group lexical rules by context like pattern-action pairs. From this point of view, we have a few remarks to emphasize, see Table 3. Another point of comparing and contrasting is the possibility of object-oriented design for these tools. It appears that Quex, ANTLR and JFlex/JLex follow an object-oriented design, but not Lex.

In addition to the above-mentioned characteristics, Quex enables visualization of the generated DFA for tokens, while ANTLR offers many facilities for the parser, including syntactic tree visualization and web implementations. JFLEX lexers depend on the defined DFA. They are quick, without costly backtracking. JFlex scanners are designed to work on text and mostly work efficiently with LALR parsers.

3.2 Some general performance guidelines

Regardless of the parser chosen, we should consider a list of heuristics that can speed up a scanner even more by using a lexical specification. They are as follows, roughly in order of performance gain:

1. Prevent rules that require backtracking methodology;

2. Eliminate line and column counting. By increasing yyline each time a line terminator is matched, most scanners can do the line counting specified in

the specification;

3. Prevent using the end-of-line operator $ and look-ahead expressions. This operator implies multiple readings of the same input, resulting in a loss of efficiency.

4. Avoid the beginning of the line operator ˆ, it costs multiple additional comparisons per match;

5. In a rule, match as much text as you can.

Although some of the aforementioned performance advice goes against the spirit of a concise and accessible specification, but it is up to the individual user to determine whether or not to apply these tactics.

In conclusion, there are advantages and disadvantages whether we choose a generator for lexical analysis or choose construction from scratch. If a generator is chosen, the main advantage is an easier and faster development, but also the disadvantage is that the lexical analyzer might be less efficient and its maintenance could be complicated. Again, to write the lexical analyzer by using a high-level language has the advantages to be more efficient and compact, but also the drawbacks of a complicated development.

3.3 ANTLR as a tool of choice

In the context of modern and efficient lexical and structured text analysis we consider ANTLR as one of the best tools currently available. Unlike most other popular tools, ANTLR uses the same mechanism for lexing, parsing, and tree parsing [14], [15], which makes it more flexible and powerful. The increase in efficiency is also due to the replacement of DFAs with LL(k) lexical analysis and the lexer produced by ANTLR is very similar to what a good handcrafted lexer would look like. It increases the speed of the lexer by replacing a state machine simulator with raw code. Other advantages of ANTLR are the possibility of executing actions during token recognition and matching of nested (recursive) structures, by using a stack. ANTLR also provides parse tree matching, which can be used for pattern matching. ANTLR syntax is easy and well documented, an important feature for developers. Furthermore, ANTLR is a really dynamic tool, as the latest version is 4.9.2 released in March 2021 [15]. As the authors mention in [15] some very popular applications like Twitter, Oracle and NetBeans IDE are using ANTLR for different tasks connected with text analysis and parsing, thus proving the qualities of this tool in this context.

4 Conclusions and discussions

Summarizing the above discussed issues of lexical analysis, some final conclusions could be drawn, as to user-defined scanners versus lexing tools. User defined scanners might be very efficient if designed by a skilled programmer or team.

Nevertheless, handcrafted scanners might be difficult to maintain if changes in the programming language are made and often cannot be reused for other programming languages or purposes. On the other hand, modern automatic frameworks and tools offer a large range of facilities. Not only do they hugely simplify the task of lexical analysis and offer support to less experienced programmers, they also allow quick testing of first programming designs. Several of them may offer Unicode support, indent anchors, word boundaries, lazy quantifiers, and performance tuning options. Also, some provide very fast and modern standalone regex libraries. In counterpart for these facilities, users might have to cope with less readable generated code and sacrifice some optimality by automating tasks. The instruments for automatic scanner should be selected carefully, in accordance with the objectives and modern, flexible tools like ANTLR should be preferred. Some issues to be taken into account nowadays should be the portability and scalability of the scanner architecture, but also the possibility of the new tools to provide support and convertibility for scanners constructed by older tools like lex or flex.

References

[1] Wang, Z., Culotta, A., *When do words matter? Understanding the impact of lexical choice on audience perception using individual treatment effect estimation*, arXiv:1811.04890v4 [cs.LG] 15 Nov 2018

[2] Gonçalves, J. Júnior, Sales, J., Moreno, M. M. and Rolim-Neto, M. L., *The Impacts of SARS-CoV-2 pandemic on suicide: a lexical analysis*, Frontiers in Psychiatry, febr. 2021 doi: 10.3389/fpsyt.2021.593918

[3] Antipas, D., Camacho-Collados, J., Preece, A. and Rogers, D., *COVID-19 and misinformation: a large-scale lexical analysis on Twitter*, Conference Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: Student Research Workshop, January 2021, doi: 10.18653/v1/2021.acl-srw.13

[4] Vírseda, R. *Learning compiler design: from the implementation to theory*, Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2 June 2021 (2021) 609-610.

[5] Vaikunta Pai, T., Jayanthila Devi, A. and Aithal, P. S. *A systematic literature review of lexical analyzer implementation techniques in compiler design*, International Journal of Applied Engineering and Management Letters (IJAEML) **4** (2020), no. 2, 285-301.

[6] Kossatchev, A. and Posypkin, M., *Survey of compiler testing methods*, Programming and Computer Software **31** (2005), 10-19.

[7] Poornima, V., Sreeram, K. and Parkavi, A., *Lexical analysis using JFLEX tool*, International Journal of Research in Engineering, Science and Management **2** (2019), no. 4, 302-303.

[8] Ortin, F., Quiroga, J., Rodriguez-Prieto, O. and Garcia, M. *Evaluation of the use of different Parser generators in a compiler construction course*, In: Information systems and technologies, WordCIST 2022, Vol. 3 (Rocha, A., Adeli, H., Dzemyda, G. and Moreira, F. eds.), Springer, 2022, 338-346.

[9] Ortin, F., Quiroga, J., Rodriguez-Prieto, O., Garcia, M. *An empirical evaluation of Lex/Yacc and ANTLR parser generation tools*, PLOS ONE. **17** (2022), no. 3. e0264326. 10.1371/journal.pone.0264326.

[10] Mogensen, T. A., *Basic of compiler design*, Department of Computer Science University of Copenhagen, 2010.

[11] Grune, D., Reeuwijk, K. van, Bal, H. E., *Modern compiler design* second edition, Springer Science+Business Media, New York 2012.

[12] Bendersky, E., *Hand-written lexer in Javascript compared to the regex-based ones*, 2013 https://dzone.com/articles/hand-written-lexer-javascript.

[13] https://www.regular-expressions.info/engine.html

[14] Pratt, T., *The Definitive ANTLR 4 Reference*, The Pragmatic Programmers, LLC, 2012.

[15] www.antlr.org