International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

A Tokenizer and Parser Generator for the Rust Programming Language

Abdullah Yousuf
Department of Computer Science
Independent Research

***Abstract*— Compilers translate source code into executable form through a sequence of stages, the first two of which are tokenization and parsing. While established tools such as Flex and Bison automate the generation of these components, they target the C programming language, which lacks the memory and thread safety guarantees that have driven widespread adoption of Rust. This paper presents a combined tokenizer and parser generator for the Rust programming language. The system accepts an input file describing tokens via regular expressions and grammar productions, then generates Rust source code implementing both a lexer (using a DFA derived from the token rules) and an LR(1) parser (using automatically generated Action and Go-To tables). Implementation challenges arising from Rust's ownership model, lack of inheritance, and absence of null and exceptions are discussed. The generator is validated through the construction of a working arithmetic calculator that respects operator precedence.**

Keywords—tokenizer; parser; compiler; Rust; finite automata; LR(1); code generation

I. INTRODUCTION

Programming languages, as a field, emerged in the 20th century due to the advent of modern semiconductor technology. Programming languages, unlike English, German, or Japanese, are special languages used to instruct a computer to perform certain tasks. Programming languages are not vague and map to processor instructions through the processes of compilation or interpretation.

Programming languages, like English, German, or Japanese, comprise of (but are not limited to) a grammar and a vocabulary. The grammar allows us to determine which sequences of “words” are valid and how they relate to other sequences of words. The vocabulary allows us to determine which sequences of “letters” are valid and their meaning.

Typically, programming languages are implemented using compilers, a purpose-built program that converts a source code file and turns it into a binary file (i.e., an executable or dynamically-linked library). At first glance, the process of converting a source code file into a binary file seems somewhat straight-forward. In actuality, the process consists of several stages including tokenization, parsing, abstract syntax tree generation, intermediate code generation, and final code generation along with several optimization stages.

This research project focuses on the first two of those stages: tokenization and parsing, which involve converting a sequence of characters into tokens and converting a sequence of tokens into an overall valid and meaningful structure through syntactic analysis, respectively.

Typically, compilers use hand-written tokenizers and parsers. The benefit of using hand-written code is that it can be customized as desired (including, but not limited to, formatting and error information). However, hand-written code takes time to write and debug. As such, it would be both tedious and cumbersome for rapidly developing projects to adopt hand-written tokenizers and parsers. In response, tools that automatically generate code for a tokenizer and parser exist, such as Flex and Bison.

The problem with these well-known tools is that they generate code specifically for the C programming language. Although C is well-known and widely used, it lacks safety guarantees, inadvertently causing many bugs. In recent years, Rust has emerged as a modern programming language that enforces safety in memory and thread contexts. These safety advantages have encouraged software projects, both new and old, to adopt Rust. However, as Rust is a relatively younger language, it lacks in tooling and libraries—including tokenizer and parser generators—that had developed over many years for the C programming language.

As such, this project aims to develop a tokenizer and parser generator for the Rust programming language in order to address the lack of similar tooling for Rust so that the compilers of the future can be written using Rust due to its advantages over C.

II. DESIGN

As mentioned in the introduction, this project focuses on building a tokenizer and parser generator. Due to the overall complexity involved building a tokenizer and parser generator, it consists of several intricate and interconnected components.

A. File Parser

The first of these components is a file parser, which takes an input file, validates it, and extracts the necessary information from it. The file parser is necessary because the application relies on a file to define what in particular to build. The file parser is the first stage in the application because the information that it extracts is used in subsequent stages of the application.

The file parser uses a technique known as recursive descent to parse input files. Recursive descent analyzes the data (be it a sequence of letters or tokens) by following a top-down approach. This

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

means recursive descent first looks at the highest-level grammatical classification and then subsequently looks at lower-level classifications in a recursive manner. The benefit of recursive descent is that it relies on recursive calls to functions for each grammatical classification. This greatly simplifies the complexity of implementing recursive descent, which is simpler than its counterpart bottom-up methods.

The application requires the file to be in a specific format as detailed in the file format documentation. The file parser ensures compliance with this format by validating all files. The usage of recursive descent aids in accomplishing this task by allowing customized error handling in response to the diverse types of invalid inputs. For example, grammatical rules in the input file must be terminated with a semi-colon on a new line following the productions in the rule. If a file contains a rule with a missing semi-colon, the file parser would detect the missing semi-colon and emit an error using recursive descent.

B. Regular Expression Parser

Input files describe the process of tokenization through rules, which associate a token with a language, which Automata Theory describes as a set of symbols drawn from a finite alphabet. According to Automata Theory, various types of languages exist with varying capabilities. The application supports the usage of all kinds of languages in token rules as long as they are classified as regular. Consequently, the application uses regular expressions to specify these languages. The benefit of using regular expressions is that they are a well-known and widely used mechanism for expressing languages.

Although more powerful languages exist, regular languages are sufficient for practical applications. This is because programming languages consist of many tokens that are fixed (i.e., expressed by a language for a single non-empty string), such as keywords.

Therefore, one of the components of the application is a regular expression parser. This takes the regular expression specified in the token rules and converts them into an automaton to be used in subsequent stages. The regular expression parser also performs validation, and in the case of invalidity, it emits an appropriate error. The regular expression parser is implemented using recursive descent as described above. Once the end of a rule is reached, the regular expression parser outputs a tree structure that is used by the non-deterministic finite automaton builder (NFA builder).

Fig. 1 shows the corresponding parse tree generated for the regular expression “(abc)+”.

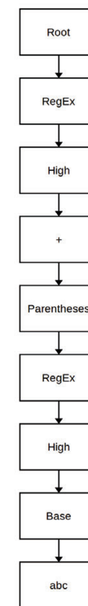

Fig. 1.
Parse tree for “(abc)+”.

C. NFA Builder

The NFA builder recurses through this tree in order to build a non-deterministic finite automaton (NFA). This is done by creating the appropriate type of automaton in the base case and connecting automata together via appropriate transitions in other cases. This process creates a data structure that accurately represents an NFA using nodes and transitions between them.

Fig. 2 shows the corresponding NFA for the regular expression “(abc)+” built from the parse tree in the previous figure.

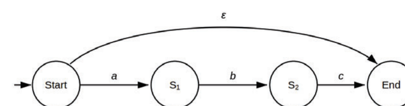

Fig. 2.
NFA for “(abc)+”. The state labeled “End” is an accepting state.

The advantages of NFAs are two-fold. For one, they can recognize regular languages, so they serve as an effective mechanism for testing the regular expression parser. Second, NFAs can be directly converted into deterministic finite automata (DFA) via an NFA to DFA conversion algorithm. The DFA is the ideal mechanism for recognizing the tokens specified due to its deterministic nature. This means that a DFA would determine whether a series of characters is in a certain language without performing backtracking or simulating multiple states at the same time. Simply put, the DFA is an

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

efficient mechanism for language recognition in terms of recognition speed.

Before converting the NFA to DFA, the NFAs for all the rules are combined into a single NFA by creating a new root node and adding a transition from it to each NFA. The reason why this is performed is because keeping track of a single NFA is easier than keeping track of multiple NFAs. Not only that, using a single NFA is ideal for the NFA-to-DFA conversion because a single DFA is obtained. On the other hand, if multiple NFAs were used, then multiple DFAs would be obtained and combining these DFAs would be cumbersome because the deterministic nature of the DFAs prohibits arbitrarily adding edges.

Fig. 3 shows an NFA generated by combining the NFA for each rule into a single NFA.

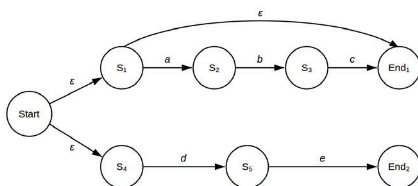

Fig. 3.
A combined NFA for the rules "(abc)+" and "de". The states End_1 and End_2 are accepting states.

D. DFA Table Conversion

The next stage of the application converts the DFA generated by the NFA to DFA conversion from a graph-based data structure into a table-based structure, which maps states to their outgoing transitions. A table-based structure is preferred for code generation because it organizes all the DFA states succinctly. This is necessary to ensure the size of the generated code is small. Beyond that, it is easier to deal with cycles in the DFA when using a table-based structure because iterating through a table can be done one state at a time whereas iterating through a graph requires traversal of the graph edges and nodes. Finally, from a theoretical standpoint, graph-based and table-based representations of a DFA are equivalent, so neither approach offers advantages in that regard.

E. Grammar Generator and LR(1) Parsing

After that, the application runs the grammar generator. Technically, the grammar generator is only subsequent to the file parser stage since it does not depend on other stages aside from that one. The grammar generator runs a series of algorithms to generate the Go-To and Action tables, which are necessary to implement the LR(1) parsing algorithm used in the application.

LR(1) is one class of parsing algorithm. It adopts a bottom-up approach in analyzing a sequence of tokens. This means it starts off by analyzing sequences for grammatical rules at the lowest level and then from there builds up to higher-level rules (which are comprised of the lower-level rules). It is deterministic in that parsing is performed by avoiding backtracking. The 1 in LR(1) indicates the parser can look up one token in the sequence.

The LR(1) algorithm was chosen to be used in the application for two main reasons. First, LR(1) parsers can handle a larger range of grammars compared to LL parsers, which employ a top-down approach. Second, LR(1) parsers depend on two tables—Go-To and Action—so generating code for an LR(1) parser would focus on generating these tables as a function of the input. This is advantageous because generating tables is easier to automate compared to generating individual functions for each grammatical rule (as would be done if recursive descent was automated).

Furthermore, some lexer and parser generators use the LALR parsing algorithm, which has a smaller-sized table compared to LR. However, LR is more powerful than LALR, and it supports a larger set of grammars. The trade-off between table size and power did not seem significant, so the author chose to use LR, which makes this project stand out among others.

F. Code Generator

The final stage of the application is the code generator, which generates the output file containing code for the tokenizer and parser for the symbols and grammar specified in the input file. The code generator writes structs and functions to the output file. It uses the table-based DFA that was created earlier to generate the functions for transitions and acceptances in the tokenizer component, and in similar fashion, it takes the Action and Go-To tables generated and writes them in a function for retrieving such tables.

III. IMPLEMENTATION

The implementation of the application focused on taking the design and writing it in the Rust programming language. As Rust is a relatively new language with unique safety features, writing the code in Rust was not met without challenges.

A. Error Handling

Many programming languages include exceptions and exception handling. However, Rust does not. Instead, Rust provides an Enum called Result, which has two subtypes Ok and Err indicating a non-error or an error. This means that every function that would potentially throw an exception instead returns a result that indicates whether an error occurred or not. Thus, the dynamic of error handling is to an extent different in Rust than other languages. The consequences of this mean that, for instance, a function written in one language could provide error handling by using one try-catch block for a series of statements whereas in Rust, error would have to be handled one potential origin site at a time.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

This combined with the lack of inheritance provided by Rust means that components of the application use a custom enum to record all the possible types of errors that occur in a particular component. In contrast to this, some notable programming languages use a custom exception class for each type of possible error. In Rust's favor, the solution adopted in the application in combination with Rust's result type requires less lines of code than creating an exception class for each possible error.

B. Graph Data Structures and Ownership

Generally, when implementing graph data structures, such as trees, a class would be used containing references to other objects of the same type. In Rust, this is not possible for two main reasons. First, Rust does not support classes, which are a key feature of many popular languages. Second, Rust as a language limits the usage of raw pointers and advocates for the use of smart-pointer types. The second point may not be a limiting factor if acyclic data structures are being implemented since the use of the smart-pointer types becomes a problem when cycles are involved.

To elaborate on the second point, Rust is strict regarding ownership and borrowing of data. This means that only a single mutable reference to a variable can exist at a time (and no other references) while multiple non-mutable references can exist concurrently (and no mutable reference). This means that creating mutable graph-based data structures requires workarounds. The first of which is using a lock within a specific smart pointer type. The problem with this approach is that adding the lock complicates the code as locks have to be opened and closed properly to avoid deadlock. The second workaround is using a raw pointer but this requires usage of an unsafe code block (i.e., a block that is labeled as unsafe). The solution offered by a workaround leads to code that is more similar to that of C, but this means some of Rust's memory features would not work. The application implements both workarounds. In practice, the second workaround, using raw pointers, was found to be easier to implement and debug due to the much-reduced complexity of the code.

C. Option Types and Null Safety

Furthermore, Rust does not support the usage of null. Instead of returning null, functions should output an Option<T> type which serves as a wrapper for the actual return type. At first glance, this seems cumbersome from the perspective of other languages. However, this design led to the application being safer from errors because functions that possibly return null are clearly marked by using Option<T> as their return type whereas functions that do not are marked with their return type being T. In addition to this, the usage of Option<T> enforces strict type checking, so before Option<T> could be used for some operation, its underlying data would have to be exposed for access in a safe way. As a result, the opportunities for errors are greatly reduced.

D. Conciseness and Code Sharing

Writing code in Rust was found to be sometimes more concise than what the equivalent code would be in other languages. For example, enums are used frequently in the application because beyond being traditional enums, they can contain data. The match expression in Rust is a powerful and concise way to handle enums in some or all of its cases. The equivalent solution in other languages would be to use if-else if statements or use a switch statement, both of which appear inferior to Rust's match mechanism.

Although the code being generated is in Rust, it is not necessary for the application to have been written in Rust to accomplish this. However, the author did find that implementing the project in Rust as well as having it generate Rust led to synergy. This is because using a common language allowed code to be shared between the generated code and the project's source code. For example, there are some structs, enums, and functions that the application uses internally and emits in its output (e.g., Action, StackSymbol, the parse function).

In addition to this, sharing code allowed debugging to be performed more easily. If a bug appeared in the generated code, the code sharing allowed the bug to be reproduced in the application itself. This led to the cause of the bug being found more easily. Thereby correcting the bug in the application also meant correcting it in the generated code due to the code sharing.

Although the generated code could be debugged, it is easier to debug the application due to a greater amount of information available (e.g., information from the algorithms that generated the parser or lexer tables). In addition to that, the generated code is not formatted perfectly due to its generated nature, and the original source code is more readable.

IV. VALIDATION

In order to test and validate the project, an example calculator program was built using the lexer and parser generator developed. The calculator program is meant to be a realistic example of a possible real-world usage of the project.

The calculator is interesting in that it involves multiple precedence levels and multiple symbols. Additionally, creating a calculator was chosen because the behavior of mathematical operations is well-defined (e.g., observing PEMDAS order) and the results are deterministic.

The calculator constitutes mathematical operations, such as addition, subtraction, multiplication, and division, as well as unary prefixes (the minus sign for negative numbers) and

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

parentheses expressions. Fig. 4 shows the grammar and token definitions of the calculator program. Fig. 5 and Fig. 6 show examples of the calculator run on two different inputs.

```
SECTION LEXER
number [0-9]+
plus \+
minus \-
times \*
divide \/
l_paren \(
r_paren \)

SECTION GRAMMAR

primary_expr: number
| l_paren expression r_paren
;

unary_expr: primary_expr
| minus primary_expr
;

term: unary_expr
| term times unary_expr
| term divide unary_expr
;

expression: term
| expression plus term
| expression minus term
;

root: expression
;
```

Fig. 4.
Input file (containing grammar and token definitions) for the calculator program.

```
Enter an expression:
-(12+-(3*4+90))
Result: 90
```

Fig. 5.
Demonstration of the calculator program.

```
Enter an expression:
1+(12-9*1/3)
Result: 10
```

Fig. 6.
Demonstration of the calculator program.

The calculator program is comprised of two separate components, the input file (i.e., the file containing grammar and tokenizer definitions) and the code file. The tokenizer and parser generator are given the input file containing the definitions and then the file containing the generated parser and tokenizer is outputted. The code file imports two functions from the outputted file, get_tokens and parse. It reads user input and passes that to the get_tokens function, then passes the output from get_tokens to parse, then finally recurses through the parse tree created by parse to calculate the result of the input.

The calculator program shows how simple it is to use the application in any general application. Other widely-used tokenizer and parser generators may be two distinct programs requiring two distinct files. Thus, it may be somewhat tedious to get these two programs to cooperate. In contrast to this, the project combines being a tokenizer generator and parser generator and generates both in a single file, so it is easier to use than other related software.

Additionally, another benefit of the project is that adding more symbols or operations to the calculator can be done easily by updating the input file and then adding code to recurse the trees for those new symbols or operations (if necessary). This is crucial in projects with high-growth as the nature of the application developed allows quick iterations over multiple designs without being committed to large amounts of code.

V. SUMMARY

The author developed a tokenizer and parser generator, which are the frontend components of compilers. While similar applications already exist, there is a gap in the availability of tokenizer and parser generators for the Rust programming language due to its relatively younger age. Rust is modern and widely-considered as a safe systems programming language, inciting many companies and projects to adopt Rust. As such, the compilers of the future would be written in Rust. Therefore, this project aims to provide a head start to new Rust-based compiler development by developing the necessary tooling.

To demonstrate the capabilities and applicability of the project, a calculator program was created using it. The nature of the project in that it combines both the tokenizer and parser under a single hood provides a consistent interface unlike other applications which separate both components. Overall, the process of creating the calculator program using the tokenizer and parser application was straightforward.

VI. FUTURE WORK

Future development on the project can expand on areas which the project did not focus on due to time and scope constraints. For instance, support for adding code directly to the input file can be implemented which would allow the user to directly generate a custom AST (or perform other tasks). The DFA minimization algorithm can be implemented to reduce the size of the tables thereby reducing the size of the generated code. Finally, support for empty productions can be added to the application to expand the range of possible grammars.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

References

[1]M. E. Lesk and E. Schmidt, “Lex — a lexical analyzer generator,” Bell Laboratories, Murray Hill, NJ, Tech. Rep., 1975.

[2]C. Donnelly and R. Stallman, Bison: The Yacc-Compatible Parser Generator. Boston, MA: Free Software Foundation, 2015.

[3]A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed. Boston, MA: Addison-Wesley, 2006.

[4]J. E. Hopcroft, R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd ed. Boston, MA: Addison-Wesley, 2006.

[5]S. Klabnik and C. Nichols, The Rust Programming Language, 2nd ed. San Francisco, CA: No Starch Press, 2023.

[6]D. E. Knuth, “On the translation of languages from left to right,” Information and Control, vol. 8, no. 6, pp. 607–639, 1965.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)