Building a Regular Expression Interpreter

solo project by Kira Velez.

Introduction and Motivation

Regular expressions (regex) are a cornerstone of modern computing, with applications ranging from data validation and syntax parsing to powering search engines and text analysis. At their core, regex provides a practical implementation of concepts from automata theory, making them a fascinating intersection of theoretical computation and real-world utility. This project focuses on creating a regex interpreter that translates regex patterns into Non-deterministic Finite Automata (NFA) using Thompson's construction algorithm. By exploring this relationship, the project aims to bridge theoretical foundations with practical software development, offering both educational insights and practical applications.

This work builds upon decades of foundational research in automata theory, particularly the seminal work of Thompson, who introduced a systematic method for constructing NFAs from regex patterns. The elegance of Thompson’s construction lies in its modular approach, breaking down regex operations like concatenation, union, and Kleene star into manageable components. Building on these ideas, this project seeks to implement a recursive descent parser and integrate it with Thompson's construction algorithm, enabling the generation and simulation of NFAs. Additionally, the project incorporates lessons from other related works on regex and automata, focusing on practical implementation challenges and computational limits.

Reiterate Goal

The primary goal of this project is to develop a regex interpreter that not only translates regex into NFAs but also demonstrates the theoretical underpinnings of automata in practical contexts. The interpreter serves as an educational tool, revealing how theoretical constructs like regular languages and finite automata operate in real-world scenarios.

Importance

Regex underpins countless computational tasks, including text searching, pattern matching, and input validation. A deeper understanding of regex and its computational limits can foster the development of more efficient tools and algorithms. Moreover, implementing such an interpreter reinforces the connection between theoretical concepts in automata and their practical applications, offering insights into the strengths and limitations of regular languages.

Difficulties

Implementing a regex interpreter involves several challenges. Handling nested structures and managing edge cases, such as invalid or incomplete regex patterns, requires a robust parser and state management system. Additionally, regex matching can become computationally expensive, especially in cases involving backtracking, where the complexity can grow exponentially. These challenges demand a careful balance between theoretical accuracy and practical efficiency.

Approach

This project employs a recursive descent parser to break down regex patterns into an Abstract Syntax Tree (AST), a structured representation of the regex's operations. Thompson's construction algorithm is then used to generate NFAs from the AST, enabling simulation of regex patterns against input strings. This modular approach ensures clarity in design and provides opportunities to expand functionality, such as supporting more complex regex features or optimizing the generated automata for better performance.

This project's timing is particularly significant, as it directly ties into coursework on automata theory and computational limits, making it an ideal opportunity to apply these concepts in a practical setting. By integrating recursive parsing and Thompson's construction, the project offers a unique blend of theoretical exploration and software engineering, with the potential to inform future studies and practical applications.

Theoretical Foundations Introduction

The methods used in this project are deeply rooted in fundamental concepts from automata theory, computability theory, and complexity theory. Automata theory provides the core framework for understanding how regular expressions (regex) correspond to finite automata. Thompson's construction algorithm translates regex patterns into Non-deterministic Finite Automata (NFA) using a structured approach based on five simple rules: empty expressions, single symbols, union, concatenation, and the Kleene star. This algorithm is a cornerstone of the project, as it systematically breaks down complex regex patterns into manageable automata components. The Chomsky hierarchy further contextualizes the regular languages generated by the interpreter, grounding the work within the broader theoretical landscape of formal languages.

From the perspective of computability theory, the project demonstrates that regex matching is a decidable problem. This is evident in the design of the interpreter, which constructs an automaton that can deterministically accept or reject input strings in finite time. By examining the reduction of regex matching to automata construction, the project illustrates how abstract

concepts like language acceptance translate into practical computational methods. Additionally, the modular design of the project—splitting regex parsing, AST construction, and NFA generation into distinct steps—reflects principles of structured computation, enabling a clear and logical workflow.

Practical Insights and Extensions

Complexity theory adds another dimension to the project, particularly in analyzing the performance of the regex interpreter. While simple patterns yield linear time complexity during simulation, more intricate patterns involving backtracking or nested operations can lead to exponential growth in execution time. This challenge highlights the importance of optimizing automaton construction and transitions. Recursive descent parsing, chosen for its alignment with regex's grammar structure, further illustrates how theoretical principles can guide practical implementation. Although the recursive approach simplifies the parsing process, it also introduces potential pitfalls, such as stack overflow in deeply nested patterns, emphasizing the need for efficient memory management and algorithmic design.

Beyond the course material, this project integrates practical insights into parser design and NFA simulation. Leveraging Python's dynamic data structures, the implementation showcases the adaptability of theoretical models in modern programming languages. The systematic application of theoretical principles to software design not only reinforces the practical utility of automata theory but also paves the way for further exploration. Enhancements like DFA conversion and optimization of automata transitions demonstrate how foundational theories can evolve into tools with real-world impact, bridging the gap between abstract computation and everyday programming challenges.

Related Work

Foundational studies, such as Thompson's work on regex-to-NFA construction and the exploration of regular languages in Hopcroft's automata theory, laid the groundwork for this project. While these provide robust theoretical insights, this project builds on their methods by implementing a parser that generates an Abstract Syntax Tree (AST) and constructs NFAs using Python, bridging gaps in practical implementation and educational tools.

Thompson's Construction Expansion

Thompson's construction algorithm is a systematic method for transforming a regular expression (regex) into a Non-deterministic Finite Automaton (NFA). This approach underpins the practical implementation of regex interpreters by providing a clear, modular framework to build NFAs

based on the structural components of a regex. The algorithm operates by decomposing a regex into atomic operations—concatenation, union, and the Kleene star—and creating corresponding NFA fragments for each. These fragments are then combined to construct a complete automaton capable of simulating the regex pattern.

Thompson's construction is guided by five core rules(Shown in Appendix A). First, an empty expression is represented by an NFA with a start state and an accept state connected by an epsilon transition. Single symbols, such as a or b, are represented similarly, with the transition between states labeled by the symbol. The union operator (|) creates a new start state with epsilon transitions to the start states of two sub-NFAs, as well as a new accept state with epsilon transitions from the accept states of both sub-NFAs. Concatenation (AB) connects the accept state of one NFA to the start state of another via an epsilon transition, effectively merging the two automata. Finally, the Kleene star (A*) adds a new start state with epsilon transitions to both the original start state and a new accept state, allowing for zero or more repetitions of the pattern.

In this project, Thompson's construction serves as the foundation for the NFA generation module. Once a regex is parsed into an Abstract Syntax Tree (AST), the tree's nodes are traversed to apply the corresponding Thompson's rules recursively. For instance, a regex like (a|b)* is first broken into its union (a|b) and Kleene star components. The union NFA is constructed by applying Rule 3 to merge the NFAs for a and b, and Rule 5 is subsequently applied to encapsulate the result within a star operation. This modular approach ensures that even complex regex patterns can be systematically translated into an NFA.

Thompson's construction not only facilitates the creation of NFAs but also demonstrates the practical connection between theoretical automata and real-world computing tasks. By following its structured methodology, this project ensures that the resulting automata accurately simulates the regex, allowing users to test input strings for matches. This implementation highlights the algorithm's versatility and reliability, while also providing an educational lens into how regex is processed at a computational level. Through the use of Thompson’s construction, the project bridges the gap between abstract theoretical concepts and their practical applications in pattern matching and language processing.

Design/Conceptual Framework

Theoretical Foundations

The design of this project is grounded in key principles from automata theory, computability, and complexity theory, which provide the theoretical framework for its implementation.

- **Automata Theory:** The core premise of the project lies in the direct mapping of regular expressions (regex) to Non-deterministic Finite Automata (NFA) components. This mapping illustrates the inherent relationship between regular languages and finite automata, as defined in the Chomsky hierarchy. By constructing NFAs from regex patterns, the project demonstrates the power and limitations of regular languages in representing computational processes.
- Computability: The project validates that regex matching is a decidable problem by implementing automata that guarantee finite execution for any given input. This property of automata ensures that the regex interpreter can reliably determine whether a string belongs to a regular language defined by a regex.
- Complexity: An important aspect of the project is the analysis of time complexity in regex matching. While simple regex patterns yield linear time complexity, more complex patterns, particularly those requiring backtracking, can exhibit exponential growth. This complexity analysis provides insights into the trade-offs between computational resources and pattern-matching accuracy.

Implementation Framework

The implementation of the project involves three major components: parsing, NFA construction, and simulation.

- Parsing: A recursive descent parser is employed to break down regex patterns into an Abstract Syntax Tree (AST). This tree provides a structured representation of regex operations, including union (|), concatenation, and the Kleene star (*). By aligning with the grammar rules of regular expressions, the parser ensures that the AST captures the hierarchical structure of the input regex accurately.
- NFA Construction: Thompson’s construction algorithm is used to systematically translate the AST into an NFA. Each operation in the regex is represented as an NFA fragment, which is then combined according to the rules of Thompson's construction. This step forms the foundation of the regex interpreter, connecting theoretical automata with practical computational models.
- Simulation: The constructed NFA is designed to simulate input strings and determine whether they match the regex pattern. The simulation leverages the properties of NFAs, including epsilon transitions, to explore all possible paths through the automaton. By testing input strings against the automaton, the project validates the correctness and completeness of the regex-to-NFA conversion process.

This structured design ensures a clear connection between the theoretical concepts and their practical implementation, offering a comprehensive exploration of regex processing through the lens of automata theory.

Completed Work

The project's development began with creating a tokenizer and parser, which are essential for converting regular expressions (regex) into an Abstract Syntax Tree (AST). The tokenizer breaks down the input regex into a sequence of tokens, such as characters, operators (|, *), and parentheses, while the parser employs a recursive descent approach to construct the AST. This structured representation of regex operations provides the foundation for subsequent processing and ensures that the hierarchical relationships between operations like union, concatenation, and Kleene star are accurately captured.

With the AST in place, the project implemented Thompson's construction algorithm to translate parsed regex into Non-deterministic Finite Automata (NFAs). Each node in the AST corresponds to a specific operation, such as union or concatenation, which is mapped to an NFA fragment according to the five fundamental rules of Thompson's construction. By systematically combining these fragments, the algorithm builds a complete NFA that represents the input regex. This implementation underscores the connection between theoretical automata and practical computation, demonstrating the feasibility of transforming abstract regex patterns into machine-readable automata.

Throughout the development process, significant effort was dedicated to debugging, particularly in handling nested expressions and mismatched parentheses. These issues posed challenges in ensuring that the parser correctly identified and managed the relationships between subexpressions. For instance, unmatched parentheses could lead to incomplete or erroneous ASTs, disrupting subsequent NFA construction. To address these problems, detailed error messages were integrated into the parser, allowing for early detection and clear communication of errors. This enhancement greatly improved the debugging process and clarified the causes of issues during development.

Comprehensive unit tests were developed for both the parser and NFA simulation to validate the correctness and robustness of the implementation. These tests included simple regex patterns, edge cases such as empty inputs and invalid syntax, and complex patterns with multiple operators and nested structures. By covering a wide range of scenarios, the test suite ensured that the components of the interpreter functioned as expected under various conditions. Additionally, these tests provided a framework for systematically identifying and resolving issues, enabling continuous improvement of the codebase.

In sum, the project's completed work demonstrates a thoughtful approach to building a regex interpreter, balancing theoretical rigor with practical implementation. The integration of robust error handling, detailed test coverage, and systematic debugging reflects a commitment to creating a reliable and maintainable tool for exploring automata theory and regex processing.

Key Findings

One of the most important insights gained from this project was the necessity of rigorous handling of edge cases in parser design. Regular expressions can include a wide range of structures, from empty inputs to invalid syntax with missing operators or mismatched parentheses. Without explicit checks and robust error handling, these edge cases can lead to parsing failures or produce invalid Abstract Syntax Trees (ASTs), which would disrupt the entire regex interpretation process. By integrating detailed error messages and validation mechanisms, the parser was able to catch such issues early, improving the overall reliability of the system and ensuring that invalid inputs were handled gracefully.

Another critical finding involved state management in Non-deterministic Finite Automata (NFAs). The construction of NFAs requires careful tracking and renaming of states, particularly when combining multiple NFA fragments during operations like union and concatenation. Errors in state naming or transitions can result in incorrect automata, leading to failed simulations or unexpected behavior. Ensuring the correctness of transitions and maintaining consistency in state identifiers were essential steps in producing functional and accurate NFAs. These findings highlight the importance of meticulous design and implementation in both parsing and automaton construction to achieve a robust and reliable regex interpreter.

Video Demonstration: Exploring the Regular Expression Interpreter

To provide a comprehensive overview of the project and its functionality, I have included a video demonstration showcasing the key features and capabilities of the Regular Expression Interpreter. The video walks through the fundamental aspects of the code, focusing on the menu-driven interface that allows users to interact with the system.

In the demonstration, I begin by navigating the main menu, which offers three primary options: parsing a regex, constructing and manipulating NFAs, and combining these features into an integrated workflow.

Parser_NFA.mp4.mp4

Accessing the Regular Expression Interpreter on GitHub

The full implementation of the Regular Expression Interpreter, including all source code, documentation, and supporting materials, is available on GitHub. The repository provides the following resources:

- **Source Code**: Contains the Python scripts for the parser, NFA construction, and menu-driven interface.
- Unit Tests: Includes comprehensive tests for the parser and NFA simulation.
- Documentation: Features a README file with detailed instructions for installation, usage, and project context.
- Video Demonstration: Offers a walkthrough of the interpreter's functionality.

You can access the repository here: git@github.com:kive7791/Parser.git

Future Work

Adding Support for DFA Construction:

Expanding functionality to include Deterministic Finite Automata (DFA) construction would significantly enhance the efficiency and practicality of the regex interpreter. Unlike Non-deterministic Finite Automata (NFA), DFAs do not require backtracking during simulation because they have a unique transition for each input symbol from any given state. This ensures that the simulation of an input string is linear in time, regardless of the regex complexity. Incorporating DFA minimization algorithms could further optimize the number of states, reducing memory usage and making the tool more suitable for applications that require high-speed, large-scale pattern matching, such as real-time search engines and data parsing tools. Additionally, DFAs serve as a powerful tool, providing students with a clear visual understanding of how complex regular expressions can be simplified into efficient deterministic models.

Enhancing Regex Syntax and Optimizing Performance:

Supporting advanced regex syntax, such as predefined character classes (e.g., \d for digits, \w for word characters) and operators like lazy quantifiers or lookahead assertions, would make the interpreter more versatile and closer in functionality to widely used regex engines like those in Python or JavaScript. This expansion would bridge the gap between theoretical automata concepts and their practical implementations in software development. Moreover, optimizing the NFA's performance by reducing state counts and streamlining transitions would improve the tool’s scalability and make it feasible for larger regex patterns. These enhancements would not

only benefit users handling complex input datasets but also showcase the real-world applicability of automata theory, highlighting its relevance in modern computational tasks. Together, these improvements would solidify the regex interpreter as both a robust educational tool and a practical utility for developers.

Team Reflection

I thoroughly enjoyed delving into Thompson's construction and its role in translating regular expressions into Non-deterministic Finite Automata (NFA). While I was able to grasp the foundational concepts, such as the five pivotal rules—empty expression, single symbol, union, concatenation, and Kleene star—the actual implementation proved to be far more complex than anticipated. The theoretical clarity provided by these rules initially helped me design the interpreter's structure, but applying them in a recursive context required an in-depth understanding that stretched beyond my current knowledge. For instance, while my implementation handled simple cases effectively, I encountered challenges with constructing NFAs for more complex patterns, such as aa*b|acb. These examples revealed gaps in my understanding of recursion and state management, which limited my ability to fully leverage Thompson's construction.

One of the most significant challenges I faced was debugging issues related to recursion and modular design. Recursive testing often led to edge cases where the construction of NFAs would break down, particularly with nested or complex combinations of regex operators. For example, while my implementation could handle basic union or concatenation, combining multiple operations in a single regex would sometimes cause errors in state transitions or lead to improperly structured NFAs. To address this, I leaned heavily on Python's features, such as its robust error-handling mechanisms and its support for dynamic data structures like dictionaries and sets. These tools were instrumental in isolating errors and refining my code, allowing me to focus on building a clear, maintainable framework for further development.

Despite these hurdles, this project provided a valuable learning experience in modular design and debugging strategies. Breaking down the regex interpreter into manageable components—such as parsing, AST construction, and NFA generation—greatly improved the clarity and maintainability of my code. By focusing on modular design, I could iterate on individual parts without disrupting the entire system. Additionally, leveraging Python's flexibility enabled me to experiment with different approaches, ultimately leading to incremental improvements in handling edge cases and simplifying recursive parsing. While there is still much room for growth, particularly in supporting more advanced regex features and improving recursive construction, this project has laid a solid foundation for future exploration in automata theory and computational problem-solving.

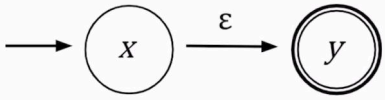
Bibliography

1. Rabin, Michael O., and Dana Scott. "Finite Automata and Their Decision Problems." *IBM Journal of Research and Development*, vol. 3, no. 2, 1959, pp. 114–125. doi:10.1147/rd.32.0114.
2. Thompson, Ken. "Programming Techniques: Regular Expression Search Algorithm." *Communications of the ACM*, vol. 11, no. 6, 1968, pp. 419–422. doi:10.1145/363347.363387.
3. "Visualizing Thompson's Construction Algorithm for NFAs: Step by Step." *Medium*, medium.com/swlh/visualizing-thompsons-construction-algorithm-for-nfas-step-by-step-f92ef378581b. Accessed Nov, 16th,2024.
4. Batra, Shalini. "Constructing NFA Using Thompson's Construction Ex I." *YouTube*, uploaded by Shalini Batra, www.youtube.com/watch?v=DryssBQeOaM. Accessed Nov, 16th,2024.

Appendix A:

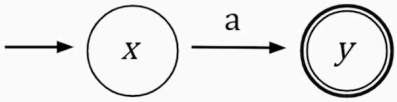
Rule #1: An Empty Expression

An empty expression, for example `''` or `ε`, will be converted to:



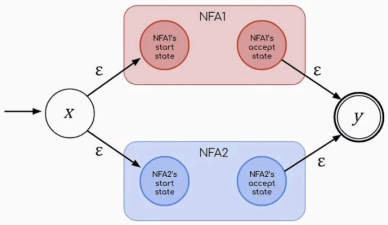
Rule #2: A symbol

A symbol, such as `a` or `b`, we will converted to:



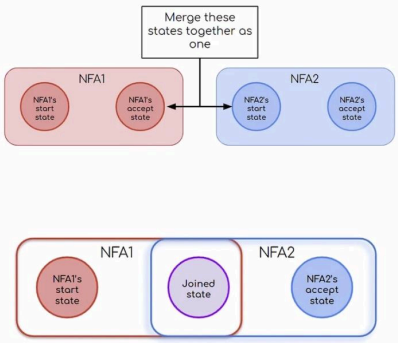
Rule #3: Union expression

A union expression `a+b` will be converted to:



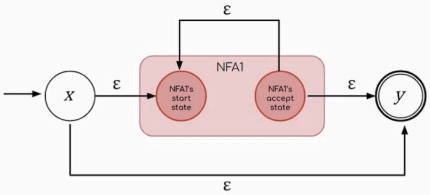
Rule #4: Concatenation expression

A concatenation expression `ab` or `a?b` will be converted to:

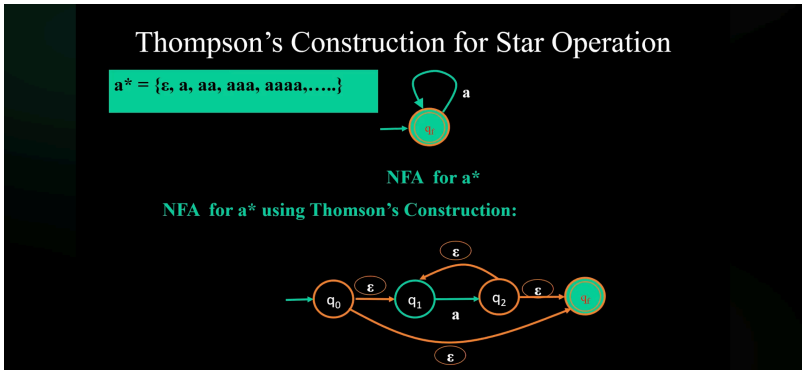


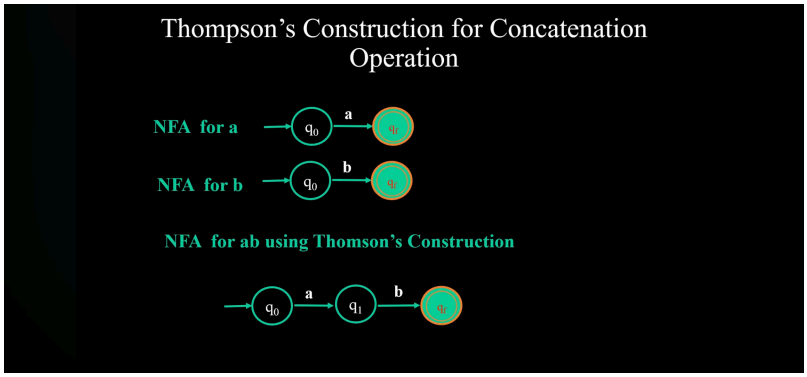
Rule #5: A closure/kleene star expression

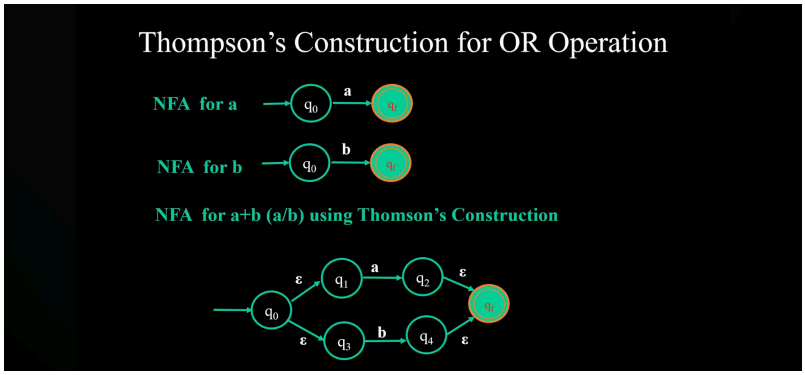
A closure `a*` will be converted to:



More examples:







Supplementary presentation if the above doesn't make sense:
https://docs.google.com/presentation/d/1EHJdxey1n07_8lcilP7_UTqXW9KFP8sHcyFARuzyEQQ/edit?usp=drive_link