# Abstract

Regular expressions (*regex*) formalize a class of patterns definable over strings, corresponding to the family of regular languages in automata theory. They provide a declarative mechanism for specifying sets of strings, making them central both to theoretical models of computation and to practical applications such as string matching, input validation, and text parsing. Despite their utility, improperly constructed regular expressions can introduce serious security vulnerabilities. One of the most critical threats is the Regular Expression Denial of Service (*ReDoS*) attack, in which carefully crafted inputs cause the regex engine to perform excessive and redundant processing. This results in dramatic slowdowns or even complete unresponsiveness of the system. ReDoS poses a significant risk to web applications, APIs, and other input-facing systems, where user-controlled input is matched against vulnerable patterns.

In this work, we propose a system to address ReDoS by transforming regular expressions into a modified *position automata*, a form of nondeterministic finite automaton (NFA) that tracks the exact start and end positions of all matches within an input string. This structure enables a matching function that computes *all* match positions, including overlapping ones, without relying on backtracking. By exhaustively and efficiently exploring the automaton's transitions, our approach avoids the exponential blowup typical of vulnerable engines, while preserving a somewhat full regex expressiveness.

Furthermore, we also review and compare this approach with existing solutions present in state-of-the-art programming languages and libraries, such as *RE#* [14] and *Hyperscan* [16].

**Palavras-chave:** regular expressions, ReDoS, position automata, nondeterministic finite automata, pattern matching.

# Resumo

O teu resumo COOL, its me TEST WOWIEESSS

**Palavras-chave:** palavra, chave..

# Acknowledgements

First of all, I would like to thank my family, etc, etc

**Dedico a . . . mim**

# Contents

# List of Tables

# List of Figures

# Listings

# Acronyms

| | |
|---|---|
| **DFA** | Deterministic Finite Automaton |
| **FCUP** | Faculdade de Ciências da Universidade do Porto |
| **NFA** | Non-deterministic Finite Automaton |
| **ReDoS** | Regular Expression Denial of Service |
| **RegEx** | Regular Expression |

# Chapter 1

# Introduction

Regular expressions are a powerful and ubiquitous tool in software development, enabling concise and expressive definitions of complex text patterns. However, their utility comes with a hidden cost: when implemented naively or misused, certain regular expression constructs can expose systems to catastrophic performance degradation. This vulnerability, known as Regular Expression Denial of Service (ReDoS), arises particularly in backtracking-based matching engines and has been responsible for real-world outages across major platforms.

This chapter introduces the ReDoS problem by first providing foundational background on regular expression engines and their operational behavior. It then outlines the nature of ReDoS vulnerabilities, illustrating their practical impact through two real-world case studies. Finally, it discusses the reasons for continued reliance on legacy engines despite known risks. By establishing the technical and practical motivations behind this research, this chapter sets the stage for the proposed approach introduced in the subsequent chapters.

## 1.1 Background

Regular expressions are a foundational tool in computer science, widely used in pattern matching, lexical analysis, input validation, and string processing. Their expressiveness and concise syntax make them a powerful language for describing regular languages.

A regular expression $R$ is used (along with an input $W$) in regex matching engines. The matching engines will verify if $W$ is fully matched by $R$, meaning that the entire input is a match - or they will verify if a substring of $W$ is matched by $R$.

## 1.2 Regular Expression Denial of Service

One such vulnerability is known as *Regular Expression Denial of Service* (ReDoS). ReDoS exploits the pathological worst-case behavior of certain regular expressions, causing exponential running time complexity during matching.

Commonly used backtracking matchers—such as those found in JavaScript, Java, and many scripting environments—ambiguous or nested expressions (such as '`(a+)+`', which involves repetition) can lead the engine to explore an exponential number of paths for certain crafted inputs. This behavior allows an attacker to intentionally supply inputs that force excessive computation, effectively rendering a service unavailable or degraded.

The root of the ReDoS problem lies not in regular expressions as a theoretical model, but in how they are implemented in many real-world software systems—particularly through backtracking-based matchers. While deterministic finite automata (DFAs) evaluate regular expressions in linear time, backtracking matchers explore multiple paths through the input, making them susceptible to exponential blow-up in the presence of ambiguous or nested patterns.

## 1.3 Case Studies

In this section, two case studies are presented. These serve as a form of introduction to getting to know and understand the ReDoS problem.

### 1.3.1 Stack Overflow

On the 20th of July 2016, a user published an malformed post on the online information exchange forum *Stack Overflow*. A couple minutes after the post, at around 14:44 UTC, the entire website became unavailable for around 34 minutes, after which there was an update that fixed the underlying issue.

On the forum, there is an automatic text formatter that runs every time someone posts something. This tool will trim any group of leading whitespace or invisible characters at both the beginning and end of a post. The regular expression that does so is the following:

$$\hat{}[\backslash s \backslash u200c] + |[\backslash s \backslash u200c] + \$$$

- ˆ will anchor the following expression to the start of a string
- $[\backslash s\backslash u200c]+$ will match one or more of either whitespace characters (space, tab, newline, etc..) or the unicode characters U+200C (zero-width non-joiner)
- \$ will anchor the match to the end of that string

The aforementioned tool was an automatic text formatter which containined a matcher that tried to match the regular expression described above against an input that contained around 20,000 consecutive whitespace characters on one line that started with $--$. The backtracking matcher in place worked as follows:

Given an input string $M$ of length $\text{len}(M)$, let $n_k$ denote the character at position $k$, where $0 \leq k < \text{len}(M)$. For each possible starting position $p$ such that $0 \leq p < \text{len}(M)$, perform the following steps:

1. Check whether the character $n_k$ (for $k \geq p$) is either a whitespace or a zero-width non-joiner (U+200C).
2. Continue checking characters until it reaches a character that is neither a whitespace nor a zero-width non-joiner.
3. If the end of the string is reached and all characters from position $p$ onward matched, the pattern succeeds.
4. If a non-matching character is encountered before the end of the string, the match fails at this position; increment $p$ and repeat from step 1.

For a 20,000 whitespace-character (both whitespace and zero-width non-joiner characters) input, the sum of computations is given as follows:

$$\sum_{k=1}^{20{,}000} k = \frac{20,000 \cdot (20,000) + 1}{2} = 200,010,000$$

This means that the matching algorithm ran in $O(n^2)$ complexity, and this blow up was to be expected.

The engineers quickly fixed this issue by switching to a substring replacing method.

### 1.3.2 minimatch

**minimatch** is a minimal matching utility, used internally by the **Node Package Manager**, better known as **npm**. [17] This utility works by convering glob expressions into JavaScript's *RegExp* objects, supporting the following glob features:

- Brace Expansion
- Extended glob matching
- "Globstar" (**) matching
- POSIX character classes

The utility has millions of downloads, as it is an essential component of **npm**.

On the 18th of October 2022, a new CVE was introduced: **CVE-2022-3517**. The report showed that all of minimatch's versions below 3.0.5 were vulnerable to a ReDoS attack similar to the one described in 1.3.1. The culprit for this was a function called *braceExpand*, which is responsible for expanding brace patterns in glob strings. This is commonly known as brace expansion and is often seen in Unix shells (such as bash). The function contained a regular expression that would match against given patterns and decide if a brace expansion was in order. The expression used was `"/\{.*\}/)"`, which matches any string containing a single pair of curly braces with any characters inside. But this expression poses an issue:

For example, the following text:

"{{{{{{{{{{{{{...X"

with the { repeated over 30,000 times and no closing }, can cause a significant CPU spike or even hang the process due to catastrophic backtracking.

To fix this issue, the developer decided to switch to a safer regular expression: \{(?:(?!\{).)*\}/

## 1.4 Reluctance Toward Changing Legacy Matchers

Despite well-documented vulnerabilities such as ReDoS, there remains significant reluctance in the software engineering community to replace or refactor legacy regex engines—particularly those built into performance-critical or widely adopted tools such as `grep`, `sed`, and many scripting languages.

These tools often rely on matching engines that prioritize speed and simplicity of implementation over safety. For example, `grep` and similar UNIX utilities implement regex matchers using finite automata, but their behavior with extended features (like bounded repetitions) can still lead to performance degradation in edge cases. While these engines are generally immune to the exponential blow-up typical of backtracking matchers, they may suffer from linear but high-cost processing when automata grow excessively large due to poorly constructed patterns.

The situation is more severe in environments that rely on backtracking matchers, such as JavaScript, Java, and many shell-based text processors. In these ecosystems, regular expressions are both expressive and dangerously permissive, allowing patterns that trigger catastrophic backtracking without warning.

Refactoring or replacing these engines is often resisted for several reasons:

- **Backwards compatibility**: Legacy codebases and systems expect specific regex semantics, and changing the underlying engine could break existing behavior.

- **Perceived performance cost**: DFA-based matchers may require significant memory or preprocessing, which is viewed as a performance risk in lightweight tools.
- **Lack of awareness**: Many developers are unaware that regex matchers can introduce denial-of-service vulnerabilities, especially when ReDoS exploits are subtle or input-driven.
- **Cultural inertia**: Tools like `grep` are deeply embedded in developer workflows and scripts, making any modification to their behavior or performance profile controversial.

Even modern matchers that are designed to avoid ReDoS—such as Google's RE2 or Rust's regex crate—are often underutilized due to these legacy constraints.

These factors highlight the need for not only technical solutions, such as safer regex engines and static analysis tools, but also a cultural shift in how regular expressions are authored, reviewed, and validated in production systems.

The next chapter will give some more insight into the basics of regular expressions and automata.

# Chapter 2

# Preliminaries

Theory builds upon theory, therefore it is essential to establish a solid foundation by understanding the basic concepts and terminology that compose the core topics of formal languages and automata theory. In this chapter we begin by formally defining what a language is and then move on to describe the class of languages known as regular languages. Along the way, we will also introduce various concepts such as finite automata (DFA, NFA) and regular expressions.

## 2.1 Alphabets, Words, and Languages

### Alphabets

An *alphabet* is a finite, non-empty set of symbols, typically denoted by the Greek letter $\Sigma$. That is,

$$\Sigma = \{a_1, a_2, \ldots, a_n\}$$

where each $a_i$ is a symbol in the alphabet.

For example, one can represent the binary alphabet as $\Sigma = \{0, 1\}$, or the English alphabet as $\Sigma = \{a, b, c, \ldots, z\}$.

### Words

A *word* over an alphabet $\Sigma$ is a finite sequence of symbols from $\Sigma$. Words are typically denoted by $w$, and the *length* of a word $w$ is denoted by $|w|$. The unique word of length zero is called the *empty word*, denoted by $\varepsilon$. The concatenation of two words $w$ and $v$ over $\Sigma$ is denoted by $wv$ (or, more explicitely, $w \cdot v$) and results in a separate word composed by the singular word $v$ right after $w$. For instance, let $w = ab$ and $v = cd$, the concatenation $wv$ is $abcd$. We can define the concatenation of two words $w$ and $v$ (which are composed of any amount of symbols) as follows:

$$w \cdot \varepsilon = \varepsilon \cdot w = w,$$
$$(wa) \cdot v = w \cdot (av), a \in \Sigma.$$

The set of all words over $\Sigma$ is denoted by $\Sigma^\star$ and is defined as:

$$\varepsilon \in \Sigma^\star,$$
$$\forall_{r \in \Sigma}, \forall_{w \in \Sigma^\star} \; aw \in \Sigma^\star$$

It is important to note that $\varepsilon \in \Sigma^*$.

For example, if $\Sigma = \{0, 1\}$, then we have that:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}$$

Where the empty word is, as mentioned above, denoted by $\varepsilon$ and also belongs to $\Sigma^*$.

Given a word $w = a_1, a_2, ..., a_n$, the reversal of $w$, denoted by $w^R$, is the word formed by reversing the order of its characters such that $w^R = a_n, a_{n-1}, ..., a_1$. We can define the reversal of a word recursively like so:

$$w^R = v^R a, \quad \text{where } w = av, a \in \Sigma \text{ and } v \in \Sigma^*,$$
$$\varepsilon^R = \varepsilon.$$

## Languages

A *language* over an alphabet $\Sigma$ is a set of words over $\Sigma$.

$$L \subseteq \Sigma^*$$

That is, a language is any subset of $\Sigma^*$, and can be either finite or infinite.
Since a language is a set of words, the following standard set operations can be applied (assuming $A$ and $B$ are languages over the same alphabet $\Sigma$):

- *Intersection*: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
- *Union*: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- *Difference*: $A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$

Furthermore, we can also operate specifically over languages with the following operations (assuming $L_1$ and $L_2$ are languages over the same alphabet $\Sigma$):

- *Concatenation*: $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
- *Kleene Star*: $L^* = \bigcup_{n=0}^{\infty} L^n$, where $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$ for $n > 0$.
- *Reversal*: $L^R = \{x^R \mid x \in L\}$, where $x^R$ denotes the reversal of a word $x$.
- *Complement*: $\overline{L} = \Sigma^* - L$, i.e., the set of all words over $\Sigma$ that are not in $L$.

These operations form the basis for reasoning about the expressiveness and closure properties of language classes such as regular, context-free, and context-sensitive languages. In this instance, the class of regular languages over an alphabet $\Sigma$ is closed under union ($\bigcup$), concatenation ($\cdot$) and Kleene star ($^*$). This robustness makes them especially amenable to algorithmic manipulation, as seen in finite automata and regular expression engines.

## 2.2 Finite Automata

A *finite automaton* is a model of computation used to recognize regular languages. It processes input words symbol by symbol and determines whether the word belongs to the language defined by the automaton. There are two main types of finite automata:

- **Deterministic Finite Automaton (DFA)**: An automaton where, for each state and input symbol, there is exactly one possible next state.
- **Non-deterministic Finite Automaton (NFA)**: An automaton that allows multiple possible transitions for a given state and input symbol, including transitions without consuming any input (called $\varepsilon$-transitions).

Formally defined, an NFA is a 5-tuple $(Q, \Sigma, \delta, Q_0, F)$ where:

- $Q$ is a finite set of states,
- $\Sigma$ is the input alphabet,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the transition function,
- $Q_0 \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of accepting (final) states.

A string $w \in \Sigma^*$ is accepted by the NFA if there exists a sequence of transitions (possibly including $\varepsilon$-moves) that consumes $w$ and ends in a state $q$ such that $q \in F$.

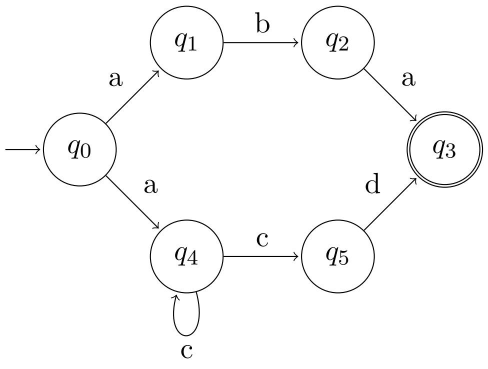

Figure 2.1: Example of an NFA that accepts $L = \{aba, a(c^*)d\}$

An NFA is *deterministic* (also known as DFA) if $|\delta(q, \sigma)| \leq 1$, for any $(q, \sigma) \in Q \times \Sigma$ and $|Q_0| = 1$.

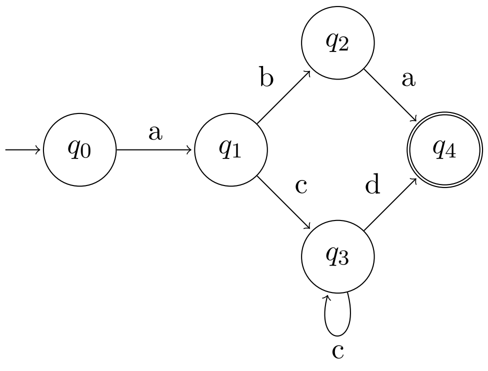

Figure 2.2: A DFA whose language is the same as the NFA from figure 2.1

## 2.3 Regular Expressions

Let $\Sigma$ be a finite alphabet. Let $L \subseteq \Sigma$. The set of *regular expressions* over $\Sigma$, denoted by RegExp($\Sigma$), is defined inductively as follows:

- $\emptyset$ is a regular expression denoting the empty language: $L(\emptyset) = \emptyset$.
- $\varepsilon$ is a regular expression denoting the language containing only the empty word: $L(\varepsilon) = \{\varepsilon\}$.
- For each symbol $a \in \Sigma$, $a$ is a regular expression denoting the singleton language: $L(a) = \{a\}$.
- If $r_1$ and $r_2$ are regular expressions, then so are:
  - $(r_1 \mid r_2)$ or $(r_1 + r_2)$, denoting the union: $L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$.
  - $(r_1 \cdot r_2)$, denoting concatenation: $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$.
  - $(r_1)^*$, denoting Kleene star: $L(r_1^*) = (L(r_1))^*$.

For each $r \in \text{RegExp}(\Sigma)$, the function $L(r)$ yields the language defined by $r$.

Parentheses are used to disambiguate expressions and enforce precedence; by convention, Kleene star binds most tightly ($a^*$), followed by concatenation (e.g. $a \cdot b$, whose operator "$\cdot$" is omitted for convenience), and finally union ($+$).

We denote by $\alpha = \beta$ if two regular expressions $\alpha$ and $\beta$, both over $\Sigma$, represent the same language ($L(\alpha) = L(\beta)$).

If the language of a regular expression $\alpha$ contains the empty word, then that regular expression possesses the *empty word property*.

**Definition 2.3.1** (Empty Word Property)**.** The empty word property is characterized by the function $\varepsilon : \text{RegExp} \to \{\varepsilon, \emptyset\}$ and is defined recursively as follows, given $\alpha$ and $\beta$ regular expressions defined over $\Sigma$:

$$\varepsilon(\emptyset) = \emptyset$$
$$\varepsilon(\varepsilon) = \varepsilon$$
$$\varepsilon(a) = \emptyset,\ a \in \Sigma$$
$$\varepsilon(\alpha + \beta) = \begin{cases} \emptyset & \text{if } \varepsilon(\alpha) = \varepsilon(\beta) = \emptyset \\ \varepsilon & \text{if } \varepsilon(\alpha) = \varepsilon \text{ or } \varepsilon(\beta) = \varepsilon \end{cases}$$
$$\varepsilon(\alpha\beta) = \begin{cases} \emptyset & \text{if } \varepsilon(\alpha) = \emptyset \text{ or } \varepsilon(\beta) = \emptyset \\ \varepsilon & \text{if } \varepsilon(\alpha) = \varepsilon(\beta) = \varepsilon \end{cases}$$
$$\varepsilon(a^*) = \varepsilon$$

To effectively compare regular expressions and discuss their equivalence in automata construction, it is useful to define a notion of similarity based on basic algebraic properties. The following definition, adapted from Brzozowski's work in [3], formalizes this concept:

**Definition 2.3.2** ([3])**.** Two regular expressions are similar if one can be transformed into the other using only the following rules:

$$\begin{aligned}
&r + r = r,\\
&p + q = q + p,\\
&(p + q) + r = p + (q + r)\\
&\varepsilon r = r\varepsilon = r\\
&\emptyset r = r\emptyset = \emptyset
\end{aligned}$$

Two regular expressions are dissimilar if they are not similar.

### 2.3.1 Extended Regular Expressions

In addition to the basic operations, some other operators are often used for convenience. These include:

- **Kleene plus**: Given a regular expression $r$, the expression $r^+$ denotes one or more repetitions of $r$:
$$L(r^+) = L(r) \cdot L(r)^*.$$
- **Fixed repetition (power)**: For a regular expression $r$ and integer $n \geq 0$, the expression $r^n$ denotes $n$ consecutive concatenations of $r$:
$$L(r^0) = \{\varepsilon\}, \quad L(r^n) = L(r) \cdot L(r^{n-1}) \text{ for } n > 0.$$
- **Bounded repetition**: For a regular expression $r$ and integers $m, n$ with $0 \leq m \leq n-1$, the bounded repetition $r^{[m,n]}$ denotes the language containing all words formed by concatenating between $m$ and $n-1$ copies of strings from $L(r)$:
$$L(r^{[m,n]}) = \bigcup_{k=m}^{n-1} L(r^k).$$

These forms do not increase the expressive power of regular expressions but are useful for readability and practical applications. They can always be rewritten using the fundamental operators: union and concatenation.

### 2.3.2 Derivatives

The *derivative of a regular expression* was first introduced in 1962 by Janusz Brzozowski. It is a powerful concept used to define the behavior of regular expressions in a more operational manner. They can be used as a means of verifying equivalence of regular expressions, for example.

**Definition 2.3.3** (Derivative of a Regular Expression [3])**.** The derivative of a regular expression $r$ with respect to $\sigma \in \Sigma$ is itself a regular expression $d_\sigma(r)$ such that $L(d_\sigma(r)) = \{w \mid \sigma w \in L(r)\}$

and is defined as:

$$\begin{aligned}
&d_\sigma(\emptyset) = \emptyset \\
&d_\sigma(\varepsilon) = \emptyset \\
&d_\sigma(r') = \begin{cases} \varepsilon & \text{if } \sigma' = \sigma \\ \emptyset & \text{otherwise,} \end{cases} \\
&d_\sigma(r + r') = d_\sigma(r) + d_\sigma(r') \\
&d_\sigma(rr') = \begin{cases} d_\sigma(r)r' & \text{if } \varepsilon(r) = \emptyset \\ d_\sigma(r)r' + d_\sigma(r') & \text{otherwise,} \end{cases} \\
&d_\sigma(r^*) = d_\sigma(r)r^* \\
&d_\sigma(r^+) = d_\sigma(r)r^*
\end{aligned}$$

This definition can be extended to a word $w \in \Sigma^*$ like so:

$$\begin{aligned}
&d_\varepsilon(r) = r, \\
&d_{wa}(r) = d_a(d_w(r)), \quad a \in \Sigma
\end{aligned}$$

With this extension:

$$L(d_x(r) = \{w \mid xw \in L(r)\}), \quad x \in \Sigma^*$$

Originally, Brzozowski defined the derivative for a concatenation of two regular expressions $r$ and $r'$:

$$d_\sigma(rr') = d_\sigma(r)r' + \varepsilon(r)d_\sigma(r'),$$

This definition will lead to an infinite number of derivatives for a regular expression. For instance, let $r = a^*$ and $x \in \Sigma^*$:

If $x = a$:

$$\begin{aligned}
d_x(r) &= d_a(a^*) \\
&= \varepsilon a^*
\end{aligned}$$

If $x = aa$:

$$\begin{aligned}
d_x(r) &= d_{aa}(a^*) \\
&= \varepsilon\varepsilon a^*
\end{aligned}$$

As seen, each of these derivatives is syntactically different and dissimilar. This pattern repeats, meaning that the derivative set is infinite. The derivatives related to the concatenation of two different regular expressions, defined in Definition 2.3.3, were proposed by Antimirov [1] and they were introduced as a way to address this issue. This modification avoids explicit multiplication by multiple $\varepsilon$ terms, ensuring that only a finite number of dissimilar derivatives are generated. Consequently, this modified definition guarantees the termination of derivative-based automaton constructions, such as Brzozowski's.

### 2.3.3 Partial Derivatives

The notion of *partial derivative* was introduced by Antimirov [1]. Contrary to Brzozowski's derivatives, *partial derivatives* will lead to the construction of an NFA.

Before defining partial derivatives, we extend the notion of concatenation to sets of regular expressions. Let $\cdot : 2^{RegExp(\Sigma)} \times RegExp(\Sigma) \to 2^{RegExp(\Sigma)}$ be a binary operation for an alphabet $\Sigma$ defined recursively as follows:

$$\begin{aligned} S \cdot \emptyset &= \emptyset, \\ S \cdot \varepsilon &= S, \\ \emptyset \cdot r' &= \emptyset, \\ \{\varepsilon\} \cdot r' &= \{r'\}, \\ \{r\} \cdot r' &= \{rr'\}, \\ (S \cup S') \cdot r' &= (S \cdot r') \cup (S' \cdot r'), \end{aligned}$$

where $S, S' \subseteq RegExp(\Sigma), r \in RegExp(\Sigma) \setminus \{\emptyset\}$, and $r' \in RegExp(\Sigma) \setminus \{\emptyset, \varepsilon\}$.

Let $r \in RegExp(\Sigma)$ be a *regular expression* over $\Sigma$. The set of partial derivatives of $r$ with respect to a symbol $b \in \Sigma$, represented as $\partial_b(r)$, is defined as follows:

$$\begin{aligned} &\partial_b(\emptyset) = \emptyset & \partial_b(r + r') &= \partial_b(r) \cup \partial_b(r'),\ r' \neq r \\ &\partial_b(\varepsilon) = \emptyset & \partial_b(rr') &= \partial_b(r)r' \cup \partial_b(r'),\ \text{ if } \varepsilon(r) = \varepsilon \\ &\partial_b(b) = \varepsilon & \partial_b(rr') &= \partial_b(r)r',\ \text{ if } \varepsilon(r) = \emptyset \\ &\partial_b(c) = \emptyset,\ b \neq c \text{ and } b \in \Sigma & \partial_b(r^*) &= \partial_b(r)r^* \end{aligned}$$

The set of partial derivatives of $r$ with respect to a word $w \in \Sigma^*$, denoted by $\partial_w(r)$, is defined recursively as:

$$\begin{aligned} \partial_\varepsilon(r) &= \{r\}, \\ \partial_{wa}(r) &= \bigcup_{r' \in \partial_w(r)} \partial_a(r'), \quad \text{for } a \in \Sigma, w \in \Sigma^* \end{aligned}$$

For instance, given the regular expression $r = (ad + d^*)db$, where $\Sigma = \{a, b, d\}$, the set of partial derivatives of $r$ for the symbol $d$ is given by

$$\begin{aligned}\partial_d(r) &= \partial_d((ad + d^*)db) \\ &= \partial_d(ad + d^*)db \cup \partial_d(db) \\ &= (\partial_d(ad) \cup \partial_d(d^*))db \cup \{b\} \\ &= (\partial_d(a)d) \cup \partial_d(d)d^*)db \cup \{b\} \\ &= (\emptyset \cup \{d^*\})db \cup \{b\} \\ &= \{d^*db\} \cup \{b\}\end{aligned}$$

resulting in $\partial_d(r) = \{d^*db, b\}$.

#### 2.3.3.1 Linear Form

It is possible to calculate the partial derivatives for every symbol $a \in \Sigma$ of a regular expression $r$ over $\Sigma$ in a single pass. To do this, Antimirov [1] defined the *linear form*.

A linear form $lf$ of a regular expression $r$ is a finite set of symbol–continuation pairs. These pairs are called *monomials* and they can encode all the possible one-symbol prefixes of that regular expression.

Formally, given an alphabet $\Sigma$ and the set of all regular expressions over $\Sigma$ denoted *RegExp*, a monomial is an element of $\Sigma \times \mathit{RegExp}$. We denote the set of all monomials for a given alphabet $\Sigma$ by $M_\Sigma$.

A *linear form* is then a finite set of monomials:

$$lf(r) \subseteq 2^{M_\Sigma}.$$

For instance, a linear form $l = \{(a_1, \alpha_1), \ldots, (a_n, \alpha_n)\}$ corresponds to the regular expression

$$\kappa_l \;=\; a_1 \cdot \alpha_1 \;+\; \cdots \;+\; a_n \cdot \alpha_n.$$

The function $lf : \mathit{RegExp} \to 2^{M_\Sigma}$ is defined recursively as follows, where $\alpha, \beta \in \mathit{RegExp}$:

$$\begin{aligned}lf(\emptyset) &= \emptyset \\ lf(\varepsilon) &= \emptyset \\ lf(a) &= \{(a, \varepsilon)\}, \quad a \in \Sigma \\ lf(\alpha + \beta) &= lf(\alpha) \cup lf(\beta) \\ lf(\alpha \cdot \beta) &= (lf(\alpha) \cdot \beta) \;\cup\; \begin{cases} lf(\beta) & \text{if } \varepsilon \in L(\alpha) \\ \emptyset & \text{otherwise} \end{cases} \\ lf(\alpha^*) &= lf(\alpha) \cdot \alpha^* \\ lf(\alpha) &= \{(a, \alpha') \mid \alpha' \in \partial_a(\alpha),\ a \in \Sigma\}\end{aligned}$$

Antimirov proved in [1, Proposition 2.5] through induction that one can decompose a regular expression into all of its one-symbol branches plus the possible empty-word case. For any regular expression $\alpha$ over $\Sigma$, the following holds:

$$\alpha = \sum_{(a,\alpha')\in lf(\alpha)} a\alpha' + \varepsilon(\alpha)$$

Equivalently,

$$\partial_a(\alpha) = \{\alpha' \mid (a,\alpha') \in lf(\alpha)\}$$

## 2.4 From Regular Expressions to Automata

While regular expressions provide a declarative way to specify patterns in words, finite automata offer an operational model for recognizing such patterns.

### 2.4.1 Thompson's Algorithm

In 1968, Thompson provided a method capable of transforming a regular expression into an equivalent nondeterministic finite automaton with $\varepsilon$-transitions (an $\varepsilon$-NFA) [13]. The main idea was to associate each regular-expression operator with a small automaton fragment, and then combine these fragments recursively until the entire expression is represented.

Each basic symbol $a \in \Sigma$ is represented by two states with a transition labeled $a$ between them. The empty word $\varepsilon$ is represented by two states connected by a single $\varepsilon$-transition.

More complex expressions are built by combining smaller fragments:

- Concatenation ($\alpha\beta$) is obtained by connecting the accept state of $\alpha$ to the start state of $\beta$ with an $\varepsilon$-transition.
- Union ($\alpha \mid \beta$) is built by adding a new start state with $\varepsilon$-transitions to the start states of $\alpha$ and $\beta$, and a new accept state with $\varepsilon$-transitions from the accept states of $\alpha$ and $\beta$.
- Kleene star ($\alpha^*$) is represented by adding a new start and accept state. The new start connects by $\varepsilon$ both to the new accept (for the empty repetition) and to the start of $\alpha$ (for one or more repetitions). The accept state of $\alpha$ connects by $\varepsilon$ back to its own start and to the new accept.

Thompson described this as a compiler-like process: the expression is first parsed into a convenient form (such as reverse Polish notation), and then a stack-based procedure builds the automaton fragment by fragment.

In the end, a single fragment represents the entire regular expression. The resulting automaton can be simulated efficiently by maintaining two lists of active states: the current list and the next list. For each input symbol, all $\varepsilon$-reachable states are added to the current list, transitions on the current symbol are followed, and the next list becomes the current list for the following step. A string is accepted if the accept state becomes active at the end of the input. An example of an automaton constructed using Thompson's method is shown in 2.3.

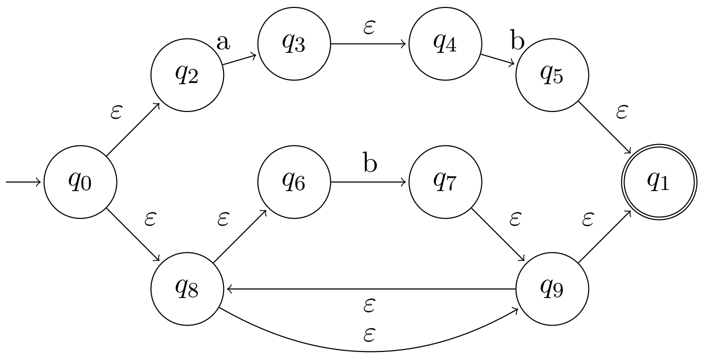

Figure 2.3: Diagram of a $\varepsilon$-NFA built using Thompson's construction for the regular expression $(ab + b^*)$.

## 2.4.2 Brzozowski's Derivatives

In 1964, Brzozowski [3] proposed a method of constructing deterministic finite automata using derivatives, where consequent derivations of a regular expression would result in the states and their transitions with respect to the symbol derived.

**Definition 2.4.1** (Brzozowski's Automaton [3])**.** Let $\alpha \in \mathbf{RegExp}$ be a regular expression defined over $\Sigma$. Brzozowski's automaton for the regular expression $\alpha$ is defined as the deterministic finite automaton $\mathcal{D} = (Q, \Sigma, q_0, \delta, F)$. $Q$ is the set of all dissimilar derivatives of $\alpha$, $q_0$ is the initial state ($q_0 \in Q$), $\delta$ is the transition function defined by $\delta : Q \times \Sigma \to Q$ such that $\delta(q, a) = d_a(q)$ for all $a \in \Sigma$ and $q \in Q$, and the set of final states is $F = \{q \in Q \mid \varepsilon(q) = \varepsilon\}$.

For instance, let $\alpha = (ad + d^*)db$ over $\Sigma = \{a, b, d\}$. The transition function $\delta$ is defined for a Brzozowski's automaton $\mathcal{D}$ of $\alpha$ as:

$$\begin{aligned}
\delta((ad+d^*)db, a) &= d_a((ad+d^*)db)\\
&= d_a(addb + d^*db)\\
&= d_a(addb) + d_a(d^*db)\\
&= d_a(a)ddb + \emptyset + \emptyset\\
&= \varepsilon ddb
\end{aligned}$$

$$\begin{aligned}
\delta((ad+d^*)db, b) &= d_b((ad+d^*)db)\\
&= d_b(addb + d^*db)\\
&= d_b(addb) + d_b(d^*db)\\
&= \emptyset + \emptyset = \emptyset
\end{aligned}$$

$$\begin{aligned}
\delta((ad+d^*)db,d) &= d_d((ad+d^*)db)\\
&= d_d(addb+d^*db)\\
&= d_d(addb)+d_d(d^*db)\\
&= \emptyset + d_d(d^*)db + d_d(db)\\
&= d_d(d)d^*db + d_d(d)b + d_d(b)\\
&= \varepsilon d^*db + \varepsilon b + \emptyset\\
&= d^*db + b
\end{aligned}$$

$$\begin{aligned}
\delta(ddb,a) &= d_a(ddb)\\
&= d_a(d)db\\
&= \emptyset
\end{aligned}$$

$$\begin{aligned}
\delta(ddb,b) &= d_b(ddb)\\
&= d_b(d)db\\
&= \emptyset
\end{aligned}$$

$$\begin{aligned}
\delta(ddb,d) &= d_d(ddb)\\
&= d_d(d)db\\
&= \varepsilon db\\
&= db
\end{aligned}$$

$$\begin{aligned}
\delta(d^*db+b,a) &= d_a(d^*db+b)\\
&= d_a(d^*db)+d_a(b)\\
&= \emptyset+\emptyset=\emptyset
\end{aligned}$$

$$\begin{aligned}
\delta(d^*db+b,b) &= d_b(d^*db+b)\\
&= d_b(d^*db)+d_b(b)\\
&= \emptyset+\varepsilon=\varepsilon
\end{aligned}$$

$$\begin{aligned}
\delta(d^*db+b,d) &= d_d(d^*db+b)\\
&= d_d(d^*db)+d_d(b)\\
&= d_d(d^*)db+d_d(db)+\emptyset\\
&= d_d(d)d^*db+d_d(d)b+d_d(b)\\
&= \varepsilon d^*db+\varepsilon b+\emptyset = d^*db+b
\end{aligned}$$

$$\begin{aligned}\delta(db, a) &= d_a(db)\\ &= d_a(d)b\\ &= \emptyset\end{aligned}$$

$$\begin{aligned}\delta(db, b) &= d_b(db)\\ &= d_b(d)b\\ &= \emptyset\end{aligned}$$

$$\begin{aligned}\delta(db, d) &= d_d(db)\\ &= d_d(d)b\\ &= \varepsilon b = b\end{aligned}$$

$$\begin{aligned}\delta(b, a) &= d_a(b)\\ &= \emptyset\end{aligned}$$

$$\begin{aligned}\delta(b, b) &= d_b(b)\\ &= \varepsilon\end{aligned}$$

$$\begin{aligned}\delta(b, d) &= d_d(b)\\ &= \emptyset\end{aligned}$$

Finally, the set of states for $\mathcal{D}$ is $Q = \{(ad + d^*)db, ddb, d^*db + b, db, b, \varepsilon, \emptyset\}$ and the set of final states is $F = \{\varepsilon\}$. The diagram for this automaton is shown in 2.4.

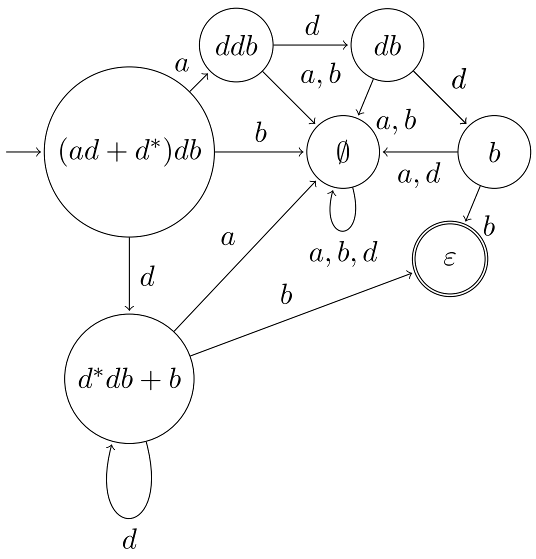

Figure 2.4: Diagram of a Brzozowki DFA for the regular expression $(ad + d^*)db$.

### 2.4.3 Antimirov's Partial Derivatives

Proposed by Valery Antimirov in 1996 [1], the partial derivatives construction generalizes Brzozowski's derivatives to build an NFA rather than a DFA. Instead of producing a single derivative for each symbol, Antimirov's method produces a *set* of partial derivatives, reflecting the inherent nondeterminism of the regular expression.

This construction avoids $\varepsilon$-transitions and yields a compact $\varepsilon$-free NFA. Each partial derivative corresponds to a transition in the automaton, and the process naturally handles alternation and repetition.

**Definition 2.4.2** (Partial Derivatives Automaton)**.** Let $\alpha \in \mathit{RegExp}$ over $\Sigma$. The partial derivatives automaton for $\alpha$ is defined as the non-deterministic finite automaton $\mathcal{N} = (Q_{\mathcal{PD}}, \Sigma, q_0, \delta, F)$. $q_0$ is the initial state and belongs to the set of states $Q_{\mathcal{PD}}$. It is important to note that $q_0 = \alpha$. The transition function is defined as $\delta(q, a) = \partial_a(q)$ for all symbols $a \in \Sigma$ and states $q \in Q_{\mathcal{PD}}$. The set of final states is defined as $F = \{q \in Q_{\mathcal{PD}} \,|\, \varepsilon(q) = \varepsilon\}$.

Recalling from 2.3.3.1, the linear form can be used as an efficient way to compute partial derivatives. Let $\alpha \in \mathit{RegExp}$ over $\Sigma$ and $a \in \Sigma$:

$$\delta(\alpha, a) = \partial_a(\alpha) = \{\beta \mid (a, \beta) \in lf(\alpha)\}.$$

That is, to compute the set of partial derivatives of $\alpha$ with respect to a symbol $a$, one inspects the linear form of $\alpha$ and collects all continuations $\beta$ that follow an $a$ in the prefix position. This characterization is not only concise but also algorithmically useful: it gives a direct recipe for building the transition relation of the partial derivatives automaton.

**Theorem 2.4.1** ([1])**.** For every regular expression $\alpha$, the partial derivatives automaton $\mathcal{N}_\alpha$ accepts exactly the language $L(\alpha)$. Moreover, the number of states of $\mathcal{N}_\alpha$ is bounded by $|\alpha| + 1$.

### 2.4.4 Position Automata

The *position automaton*, also known as the *Glushkov automaton*, is a $\varepsilon$-free nondeterministic finite automaton (NFA) constructed directly from a regular expression. Unlike the standard Thompson construction, which introduces $\varepsilon$-transitions that must later be eliminated, the Glushkov construction yields an automaton in which each state corresponds uniquely to a symbol occurrence—or *position*—in the expression. [2]

Given a regular expression $r$, the Glushkov automaton $M_r$ is defined based on four key position-based functions:

- **first**$(r)$: the set of positions that can appear first in some word of the language $\mathcal{L}(r)$.
- **last**$(r)$: the set of positions that can appear last in some word of $\mathcal{L}(r)$.

- **follow**$(r, x)$: for each position $x$, the set of positions that can immediately follow $x$ in some word of $\mathcal{L}(r)$.
- **symbol**$(x)$: for each position $x$, symbol$(x)$ represents the symbol on that position.

To distinguish different occurrences of the same symbol, the construction introduces marked symbols. For example, the expression $(a+b)^*a(b+a)^*$ is rewritten as $(a_1+b_2)^*a_3(b_4+a_5)^*$, where the positions are $\{1,2,3,4,5\}$. Each marked symbol corresponds to a unique position and becomes a distinct state in the automaton.

The Glushkov automaton $M_r = (Q, \Sigma, \delta, q_0, F)$ is constructed as follows:

- $Q$ is the set of positions in $r$ (i.e., the marked symbols), plus an initial state $q_0$.
- For each symbol $a \in \Sigma$:
  - $\delta(q_0, a) = \{x \in \text{first}(r) \mid \text{symbol}(x) = a\}$
  - $\delta(x, a) = \{y \in \text{follow}(r, x) \mid \text{symbol}(y) = a\}$
- The set of final states $F$ is last$(r)$; if $\varepsilon \in \mathcal{L}(r)$, then $q_0$ is also final.

This automaton captures the structural flow of $E$ by tracing symbol sequences as state transitions. It is $\varepsilon$-free and has one state per symbol occurrence, which results in at most a quadratic number of transitions with respect to the size of $E$.

An important property of the Glushkov automaton is its relationship to unambiguity. A regular expression is *weakly unambiguous* if and only if its Glushkov automaton is unambiguous, i.e., there exists at most one accepting path for each accepted word. This makes the Glushkov construction a practical and efficient tool in applications requiring unambiguous parsing, such as syntax analysis in document type definitions (e.g., SGML and XML DTDs).

## 2.5 FAdo

The FAdo [12] project is an open-source implementation of several sets of tools for formal languages manipulation. In order to allow quick prototyping and testing of algorithms, these tools were developed in Python. Regular languages can be represented by regular expressions which are defined in `reex.py` - or by finite automata which are defined in `fa.py`.

### 2.5.1 Finite Automata

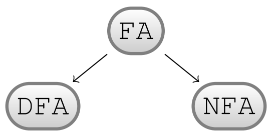

Figure 2.5: Class inheritance organization in FAdo's finite automata code file, `fa.py`.

The class `FA` defines the abstract interface for finite automata, introducing the fundamental components: the set of states, the alphabet, the transition function, the initial state(s), and the set of final states. This class also establishes common methods for manipulating these elements and for interoperability between automata.

Two subclasses will then inherit from `FA`:

- `NFA` — implements nondeterministic finite automata, possibly with $\varepsilon$-transitions. It provides methods for epsilon-closure, elimination of $\varepsilon$-moves, subset construction into a DFA, and regular operations such as union, concatenation, and Kleene star.
- `DFA` — implements deterministic finite automata. It enforces determinism by construction and provides efficient word evaluation and minimization routines.

### 2.5.2 Regular Expressions

In `reex.py`, FAdo defines the `RegExp` base class for all regular expressions, declaring therein a class variable `sigma`, formally known as the set of symbols ($\Sigma$). To represent any and all base constructions of a regular expressions, FAdo defines base classes for each of them:

- **CEmptySet**: The empty symbol set. ($\emptyset$)
- **CEpsilon**: The empty word. ($\varepsilon$)
- **CAtom**: A simple symbol. (e.g. $'a'$)
- **CDisj**: The $+$ operation between symbols. (e.g. `CDisj(CAtom(a),CAtom(b))` represents the regular expression $r = a + b$ where $a, b \in \Sigma_r$)
- **CConcat**: The $\cdot$ operation
- **CStar**: The Kleene closure over a set of symbols (e.g. `CStar(CDisj(CAtom(a),CAtom(b)))` represents the regular expression $r = (a + b)^*$ where $a, b \in \Sigma_r$)

In order to parse expressions into FAdo's classes and types, *lark* was used. *lark* is a parsing toolkit for Python. It can parse all context-free languages.

# Chapter 3

# State of the Art

## 3.1 Introduction

Regular expressions (regex) remain one of the most powerful and widely adopted tools for string pattern matching across programming languages, search tools, and data processing pipelines. While their theoretical foundation lies in formal language theory, the practical implementation of regex engines often diverges from the idealized models. This chapter mentions the state of the art in regex engine designs and regex language features, with a particular focus on performance trade-offs, security concerns, and evolving capabilities.

## 3.2 Regex Syntax Variants

Beyond the theoretical framework of regular expressions lies a diverse set of practical syntax variants. Different tools, programming languages, and libraries adopt different interpretations and extensions of regex syntax, leading to a rich but inconsistent ecosystem. Two of the most influential and widely adopted syntax families are POSIX and PCRE.

### 3.2.1 PCRE: Perl-Compatible Regular Expressions

PCRE (Perl-Compatible Regular Expressions) define an expressive and flexible regex syntax, modeled after the pattern language used in the Perl programming language. It has become the standard for regular expression syntax in many modern programming environments, including PHP, Python, JavaScript, and others.

PCRE introduces a wide range of features that go far beyond the capabilities of POSIX regexes. These include:

- **Backreferences**, allowing a pattern to refer to previously matched groups.

Example: `(a)b\1` matches `aba` (the `\1` refers to the first captured `a`).

- **Lookahead and lookbehind assertions**, enabling matches based on surrounding context without consuming it.
  Examples:
  - Lookahead: `foo(?=bar)` matches `foo` only if followed by `bar`.
  - Lookbehind: `(?<=foo)bar` matches `bar` only if preceded by `foo`.
- **Non-greedy quantifiers**, which match as little as possible.
  Example: `<.*?>` matches the shortest HTML-like tag, such as `<b>` in `<b>bold</b>`, instead of greedily matching the entire string.
- **Named capture groups**, for improved readability and maintainability.
  Example: `(?<year>\d{4})-(?<month>\d{2})` captures `2025-09` with named groups `year` and `month`.
- **Conditional expressions**, recursion, atomic groups, and other advanced constructs.
  Examples:
  - Conditional: `(a)?b(?(1)c|d)` matches `abc` if `a` is present, `bd` otherwise.
  - Recursion: `(a|b(?1))*` matches balanced nested patterns using self-reference.
  - Atomic group: `(?>a+)` prevents backtracking inside the group.

The added power of PCRE comes at the cost of increased complexity in both the syntax and the engine. To support these features, most PCRE-compatible engines rely on backtracking-based evaluation, which explores all potential matching paths through a pattern. While this approach enables support for advanced constructs, it also introduces the risk of exponential-time behavior in certain patterns—especially those involving nested quantifiers or ambiguous alternations.

### 3.2.2 POSIX Regular Expressions

The POSIX (Portable Operating System Interface) standard defines a family of regular expression syntaxes intended to promote consistency across UNIX-like systems. POSIX defines two primary levels of syntax: *Basic Regular Expressions* (BRE) and *Extended Regular Expressions* (ERE). These are used in many command-line utilities such as `grep` [6], `sed` [7], and `awk` [8].

In BRE, several metacharacters — such as parentheses, curly braces, and the plus sign — must be escaped to be interpreted as operators. For example:

- `a\{3\}` matches exactly three consecutive `a`’s (`aaa`, for instance).
- `\(a|b\)` matches either `a` or `b` using grouping and alternation.

In contrast, ERE relaxes these constraints and allows a more expressive and readable syntax without requiring escapes for most metacharacters. For example:

- `a{3}` matches `aaa` (three consecutive `a`'s).
- `a+` matches one or more `a`'s.
- `a|b` matches either `a` or `b`.
- `(abc)?` matches either `abc` or $\varepsilon$.

A defining feature of POSIX regular expressions is their *leftmost-longest match* semantics. When multiple matches are possible, the engine must return the one that starts at the earliest position in the input and is the longest among those starting at that position. This requirement ensures deterministic behavior but can increase the complexity of the matching algorithm. [10]

POSIX regular expressions do not support many advanced features common in other regex dialects. For example, they lack support for backreferences, lookahead/lookbehind assertions, non-greedy quantifiers, and conditional expressions. This makes POSIX regexes more limited in expressiveness but often easier to reason about and implement efficiently.

## 3.3 Engine Architectures

Regular expressions are most commonly used to look for patterns of strings, known as matching. The basis for this operation are the regular expression engines. At a high level, regex engines can be grouped by how they traverse the implicit nondeterministic automaton of a pattern. Backtracking engines explore one path at a time, pushing alternative choices on a stack; NFA simulation engines keep all active states in parallel; DFA and hybrid engines trade memory for predictable time; and high-throughput engines emphasize vectorization and streaming.

### 3.3.1 Backtracking NFA

The classic backtracking NFA matcher is driven by the pattern: the regular expression acts like a small procedural program that dictates how the engine explores matches and handles failure. The engine begins at the start of the text and attempts to match the pattern from that position; if it fails, it "bumps along," advancing one character and trying again. Once the first tokens of the pattern match the text, the engine proceeds through the regex. At any choice point—such as an alternation, an optional, or a quantifier—it selects one alternative to try and records the others, together with the current input position. If the chosen path later fails, the engine backtracks to the most recent choice point and resumes with a saved alternative. If all saved alternatives are exhausted, the attempt from that starting position fails and the engine bumps along to the next character.

If the engine reaches the end of the pattern with all constraints satisfied, it declares success and discards any remaining, unexplored alternatives. A key consequence is that the order of alternatives matters: typical backtracking engines implement leftmost-first semantics rather than leftmost-longest, so the first workable alternative can win even if a longer match exists.

Because the engine literally follows the structure of the regex, you control its search by how you write the pattern: place safer or more selective alternatives first, avoid ambiguous constructs under repetition, and structure patterns to minimize backtracking and to fail fast when appropriate.

Despite the name, a backtracking NFA engine is not an NFA in the formal sense used in automata theory. Theoretical NFAs are memoryless machines that recognize only regular languages, whereas real-world backtracking engines—such as those used in PCRE, Java, and Python—go beyond regularity (Câmpeanu et al. proved this using a pumping lemma in [4]). They simulate nondeterministic behavior by exploring multiple paths through the pattern, but they do so using a stack and internal memory to track backtracking points, group captures, and even previous input matches. This allows them to support powerful features like backreferences and conditional expressions, which cannot be recognized by finite automata. The term "NFA" here refers to the engine's matching strategy, inspired by the branching behavior of NFAs, rather than to its computational limitations.

### 3.3.2 POSIX NFA

Similarly to the classic NFA engine, the POSIX NFA engine will match in the same way but memorize and continue when a successful match is found. This is done to see if a longer, leftmost match can be found later on.

### 3.3.3 DFA

A DFA-based matcher is driven by the input text rather than the structure of the regular expression. The pattern is first compiled into a deterministic finite automaton (DFA), which enables the engine to scan the input in a single pass. As each character is read, the engine transitions deterministically from one state to another, maintaining only a single active state at any point during execution.

Each DFA state represents a set of possible continuations in the pattern, allowing the matcher to recognize valid strings without the need for backtracking. This results in highly predictable performance, with matching taking linear time in the length of the input.

In search mode, DFA-based engines typically return the leftmost match. Many implementations are also designed to return the leftmost-longest match by continuing the scan while remembering the last encountered accepting state. Because the DFA encodes all matching possibilities in advance, the matching process is entirely deterministic and independent of the syntactic order of alternatives in the pattern.

One practical consideration in DFA-based matching is the potential growth of the automaton during the compilation phase. In the worst case, the number of DFA states can grow exponentially with respect to the size of the regular expression, particularly for patterns involving complex nesting or alternation. While this does not affect runtime performance during matching — which remains linear — the memory and time costs of building the DFA can be significant. To address this, many modern implementations adopt strategies such as lazy DFA construction or hybrid evaluation models that balance determinism with scalability.

### 3.3.4 Hybrid

Hybrid engines try to take the best of both worlds in both NFAs and DFAs. They perform a depth-first search through the space of matches. At each point of nondeterminism—due to alternation, optional constructs, or unbounded quantifiers — they choose one option to continue and save the others as choice points on an internal stack. If a later step fails, the engine pops a choice point and resumes from there. This approach yields intuitive behavior and supports rich features, including capturing groups, backreferences, and look-around. However, in the presence of ambiguous subpatterns under repetition, the number of explored paths can grow exponentially, making these engines particularly susceptible to ReDoS.

Spencer's classic backtracking implementation is perhaps the best known and most used, embodying closely the architecture described above. Conceptually, its state comprises the current regex node, the current input index, and a stack of saved alternatives. Consider the pattern `(a|aa)+$` against the input `aaaa...a` without a trailing `b`. The matcher repeatedly chooses the left alternative `a`, consuming a single character while saving a choice point for the `aa` branch at each iteration. At the end of input it fails to satisfy the end anchor, so it begins to backtrack, popping choice points and trying to repartition the run of `a`s using `aa` segments. The number of such partitions grows exponentially with the length of the input, and the engine must examine many of them before determining there is no match. This example illustrates how overlapping alternatives inside a quantifier, combined with a terminal failure, can lead directly to catastrophic backtracking. The tools GNU egrep and awk use a hybrid approach for this. They choose

## 3.4 Engines and Libraries

### 3.4.1 RE2

RE2 is a regular expression engine developed by Russ Cox at Google, designed to provide predictable performance and strong safety guarantees. Unlike backtracking engines, RE2 avoids features that lead to exponential-time behavior, such as backreferences and arbitrary lookbehinds. By restricting itself to regular languages, RE2 ensures that all matches can be performed in linear time. [5]

The engine compiles patterns into automata using a combination of DFA and NFA techniques. For simple match queries, RE2 prefers deterministic finite automata (DFA) for their speed and predictability. When submatch extraction is needed, it may fall back to an NFA simulation, provided the regex satisfies certain properties (e.g., being one-pass). [5]

By trading off advanced features like backreferences for performance and safety, RE2 provides a robust alternative to PCRE-style engines, especially in environments where worst-case behavior must be avoided.

RE2 can also operate in either POSIX mode (accepting POSIX syntax regular expressions) or Perl mode (accepting PCRE syntax regular expressions). [9]

### 3.4.2 PCRE2

PCRE2 is a modern and widely adopted regular expression library that implements the rich, feature-complete syntax mentioned above (PCRE). [11]

At its core, PCRE2 relies on a backtracking-based matcher, which simulates nondeterminism by exploring alternative execution paths through the pattern. Although this resembles the behavior of nondeterministic finite automata (NFAs), it does not correspond to an NFA in the theoretical sense. Instead, the engine maintains an explicit control stack and memory for backtracking, enabling it to handle features like backreferences and complex group captures. This matching strategy enables high expressiveness but comes with the risk of exponential time complexity in certain pathological cases, particularly when ambiguous repetition and nested alternations are involved. [11]

PCRE2 also offers an alternative, limited matching mode based on deterministic finite automata (DFA), which guarantees linear-time performance but does not support the full range of Perl-compatible features. Despite these limitations, PCRE2 remains widely used in scripting languages and developer tools due to its flexibility and familiar syntax. [11]

### 3.4.3 Hyperscan

Hyperscan is Intel's regular expression matching engine, designed specifically for high-throughput and low-latency applications. It serves as a core component in several security and networking tools, including intrusion detection systems like Suricata and firewalls.

Traditional regex engines (e.g., backtracking-based ones like PCRE) often struggle with performance bottlenecks due to their sequential nature and vulnerability to ReDoS attacks. Hyperscan addresses these limitations by combining multiple automata models—particularly NFAs and DFAs—with a hybrid execution strategy that leverages Single Instruction, Multiple Data (SIMD) parallelism and tiled execution on modern CPUs .

According to Wang et al. ([15]), Hyperscan divides regexes into multiple subgraphs, such as anchored DFAs for simple patterns and NFAs for complex constructs. This hybrid approach enables it to process large volumes of data streams efficiently without the exponential-time risks associated with backtracking engines.

However, as noted in [15], Hyperscan will also enforce syntactic restrictions when compiling. Regexes that are deemed vulnerable or considered ambiguous, therefore limiting expressiveness and versatility, especially when the user is looking for nested repetition (e.g. $(a+)+$, looking to match one or more of one or more $a$'s) or greedy alternation (e.g. $(a|aa)+$, matching a sequence of $a$ or $aa$, repeated, resulting in three matches for a string such as $aa$).

### 3.4.4 Rust's `regex` Crate

Implements a hybrid DFA/NFA model and guarantees linear-time performance by excluding features like backreferences.

## 3.5 Prevalency of ReDoS and solutions

### 3.5.1 Revealer

Mention the revealer paper here!

## 3.6 Reluctance

If it works...

# Chapter 4

# Counting

In this chapter, we will explore the concept of counting in the context of formal languages and automata theory as well as explain an attempt that was made towards match counting using a partial derivative automaton construction and why it didn't work.

## 4.1 Counting in Formal Languages

In formal language theory, counting refers to the ability of a language or an automaton to enforce numeric constraints over the number of symbols or patterns within strings. Specifically, it deals with the ability to recognize whether certain elements occur a specified number of times—or in a specific numerical relationship to others. Current tools are already able to do this, including some non-backtracking matchers.

## 4.2 Derivatives of operations in Extended Regular Expressions

Given a word $w = xyz$ and a regular expression $\alpha$, $w \in L(\alpha)$ if $\varepsilon(D_w(\alpha)) = \varepsilon$. In order to match with fixed and bounded repetition, one must first solve the derivatives for those operations.

### 4.2.1 Fixed Repetition

For a fixed repetition, we need $r$ to occur $n$ times, resulting in the notion $r^n$.

Given $r = ab$ such that $r^2 = abab$, we have:

$$\begin{aligned} D_a(r^2) &= D_a(r \cdot r) \\ &= D_a(abab) \\ &= D_a(a)bab + D_a(bab) \\ &= \varepsilon bab + \emptyset \\ &= bab \end{aligned}$$

### 4.2.2 Bounded Repetition

Supplementing the fixed repetition notion, we now add a maximum bound so that $r$ can occur at least $n$ times and at maximum $m-1$ times: $r^{[n,m[}$.

Given $r = ab$ such that $r^2 = abab$ and $r^3 = ababab$, we have:

$$\begin{aligned} D_a(r^{[2,4[}) &= D_a(abab) + D_a(ababab) \\ &= D_a(abab) + D_a(ababab) \\ &= D_a(a)bab + D_a(bab) + D_a(a)babab + D_a(babab) \\ &= \varepsilon bab + \emptyset + babab + \emptyset \\ &= bab + babab \end{aligned}$$

This also extends for $r^{[m,\text{inf}[}$, which is the same as having $r^m \cdot r^*$:

$$\begin{aligned} D_a(r^{[2,\text{inf}[}) &= D_a(abab) + D_a((ab)^*) \\ &= D_a(a)bab + D_a(bab) + D_a(ab) \cdot (ab)^* \\ &= \varepsilon bab + \emptyset + (D_a(a)b + D_a(b)) \cdot (ab)^* \\ &= bab + (\varepsilon b + \emptyset) \cdot (ab)^* \\ &= bab + b(ab)^* \end{aligned}$$

The regular expression $r^{[2,\text{inf}[}$ will match any $(ab)^k$ with $k \geq 2$.

## 4.3 Class implementation in FAdo

In order to implement these new syntactic constructs of extended regular expressions in FAdo, we added exact-power and counted-repetition nodes to the regular expression abstract syntax tree and define their derivatives, partial derivatives and linear form cases.

### 4.3.1 CPower

*CPower* is the class responsible for the construction of regular expressions using the fixed repetition operator.

```
class CPower(Unary):
   def __init__(self, arg, n, sigma=None):
   self.arg = arg
   self.n = n
   self.Sigma = sigma
   self._ewp = False if self.n > 0 else True

   ...
```

As shown above, the class inherits from the *Unary*, which only defines the *Unary.Sigma* and *Unary.arg* properties, letting the class hold a alphabet set (representing Σ) and the symbol used to construct this operator.

In order to integrate the class fully with *FAdo*, some more methods had to be implemented.

```
   def linearForm(self):
      arg_lf = self.arg.linearForm()
      lf = dict()
      for head in arg_lf:
         lf[head] = set()
         for tail in arg_lf[head]:
            if self.n == 0:
               lf[head].add(CEmptySet(self.Sigma))
            elif self.n == 1:
               lf[head].add(CEpsilon(self.Sigma))
            else:
               lf[head].add(CPower(self.arg, self.n-1, self.Sigma))

      return lf

   def partialDerivatives(self, sigma):
      return self.arg.partialDerivatives(sigma)

   def derivative(self, sigma):
      if str(sigma) in str(self.arg):
         if self.n == 0:
            return CEmptySet(sigma)
         elif self.n == 1:
            return self.arg.derivative(sigma)
         else:
            return CConcat(self.arg.derivative(sigma), CPower(self.arg,
                self.n-1, self.Sigma))
      else:
         return CEmptySet(sigma)
```

## 4.4 Extended Regular Expressions in FAdo

Recalling back from 2.3.1, in order to support extended regular expressions, one needs to extend the base class for unary operations in FAdo

# Chapter 5

# Matching

*Matching* is the process of checking whether a piece of text fits a specific pattern described using a regular expression (regex). In this chapter, we will discuss the different approaches to matching regular expressions, the implications of using them, and the performance considerations that arise from these choices. Furthermore, we will also present a novel approach based on a modified position automata, which aims to mitigate the performance issues associated with traditional regex engines while preserving some of the extended expressiveness of regex patterns.

## 5.1 Overlapped versus Non-Overlapped Matching

In the context of regular expression matching, two distinct paradigms exist: *overlapped matching* and *non-overlapped matching*. Understanding their differences is crucial when designing matching engines, especially when completeness or performance is a concern.

### Overlapped Matching

Overlapped matching refers to finding all possible matches of a pattern in an input string. This is typically achieved by attempting a match starting at every index of the input. It is more exhaustive and useful in domains where no potential match should be missed, such as bioinformatics (DNA pattern searching).

For example, the indexed string $w = a_0a_1a_2a_3$, when matched against using the pattern $aa$, will yield the following matched substrings:

- $w_{[0,2[} = \textcolor{red}{aa}aa$
- $w_{[1,3[} = a\textcolor{red}{aa}a$
- $w_{[2,4[} = aa\textcolor{red}{aa}$

### Non-Overlapped Matching

Non-overlapped matching (also referred to as *disjoint*, *standard*, or in some engines, *greedy* matching) finds matches sequentially from left to right, and once a match is found, it advances the input pointer beyond the match. Unlike overlapped matchers, where overlapping is a feature by default, greedy matchers will often depend on the lookahead assertions to do so.

Using the same pattern $aa$ on $w = a_0a_1a_2a_3$, a non-overlapped matcher may return:

- $w_{[0,2[} = aaaa$
- $w_{[2,4[} = aaaa$

To summarize, overlapped matching provides a more complete view of potential matches, but at a higher computational cost. It is particularly well-suited to automata-based approaches like the modified position automaton described in this work.

## 5.2 Modified Position Automata

A *position automaton* is a type of nondeterministic finite automaton (NFA). We can enable overlapped matching by modifying the position automaton's construction using the algorithm described on 1.

**Algorithm 1** NFAPOSCOUNT($R$): Construct Special Position Automaton

```
Require: Regular expression $R$
Ensure: NFA $A$
 1: $A \leftarrow$ new empty NFA
 2: $i \leftarrow A$.addInitialState()
 3: $A$.addTransitionStar$(i, i)$                          ▷ Accept any symbol from $\Sigma$
 4: $f_R \leftarrow R$.marked()
 5: stack $\leftarrow$ empty stack
 6: addedStates $\leftarrow$ empty map
 7: for all $p \in First(f_R)$ do
 8:     $q \leftarrow A$.addState$(p)$
 9:     addedStates$[p] \leftarrow q$
10:     stack.push$((p, q))$
11:     $A$.addTransition$(i, p, q)$
12: end for
13: while stack is not empty do
14:     $(s, s_{\text{idx}}) \leftarrow$ stack.pop()
15:     for all $t \in Follow(f_R, s)$ do
16:         if $t \in$ addedStates then
17:             $q \leftarrow$ addedStates$[t]$
18:         else
19:             $q \leftarrow A$.addState$(t)$
20:             addedStates$[t] \leftarrow q$
21:             stack.push$((t, q))$
22:         end if
23:         $A$.addTransition$(s_{\text{idx}}, t, q)$
24:     end for
25: end while
26: $e \leftarrow A$.addState()
27: $A$.addTransitionStar$(e, e)$
28: for all $p \in f_R$.Last() do
29:     if $p \in$ addedStates then
30:         $A$.addFinal(addedStates$[p]$)
31:         $A$.addTransitionStar(addedStates$[p], e)$
32:     end if
33: end for
```

## 5.3 Automata-Based Matching

Given a regular expression $R$, one can construct an NFA $A$ such that $L(A) = L(R)$. Matching then reduces to verifying whether the automaton $A$ accepts the input string $s$. In DFA-based engines, each character of the input leads to a deterministic transition from one state to another, resulting in a guaranteed linear-time match. In contrast, NFA-based engines may involve branching paths due to nondeterminism and can require simulating multiple transitions concurrently.

For example, consider the following regex pattern:

```
^(a+)+$
```

## 5.4 Matching with the Modified Position Automaton

One can find all matches over an input string by constructing the modified position automaton from a regular expression and then simulating the automaton's transitions over the input string. The algorithm presented in this section is designed to track the start and end positions of all matches, including overlapping ones, without relying on backtracking.

**Algorithm 2** TABLEMATCHER$(A, s)$: Modified Position Automaton Multi-matcher

```
Require: $A = (\Sigma, Q, \delta, I, F)$: NFA
Require: $s$: input string
Ensure: $M$: mapping from final states to lists of match positions
 1: $symbols \leftarrow$ list with $\varepsilon$ prepended to $s$
 2: $currentRow \leftarrow$ empty map from states to list of position pairs
 3: $finalMatches \leftarrow$ empty map from states to list of matches
 4: $position \leftarrow 0$
 5: for all $sym$ in $symbols$ do
 6:     if $sym = \varepsilon$ then
 7:         for all $q_0 \in I$ do
 8:             $currentRow[q_0] \leftarrow [(0, 0)]$
 9:         end for
10:     else
11:         $nextRow \leftarrow$ empty map
12:         if $sym \in \Sigma$ then
13:             for all $q \in$ keys$(currentRow)$ do
14:                 if $|\delta(q, sym)| > 0$ then
15:                     for all $q' \in \delta(q, sym)$ do
16:                         for all $(start, \_) \in currentRow[q]$ do
17:                             if $q' = q$ and $q' \in I$ then
18:                                 append $(position, position)$ to $nextRow[q']$
19:                             else
20:                                 append $(start, position)$ to $nextRow[q']$
21:                                 if $q' \in F$ then
22:                                     append $(start, position)$ to $finalMatches[q']$
23:                                 end if
24:                             end if
25:                         end for
26:                     end for
27:                 end if
28:             end for
29:         else                                    ▷ Symbol not in $\Sigma$; treat as fresh start
30:             for all $q_0 \in I$ do
31:                 $nextRow[q_0] \leftarrow [(position, position)]$
32:             end for
33:         end if
34:         $currentRow \leftarrow nextRow$
35:         $position \leftarrow position + 1$
36:     end if
37: end for
38: return $finalMatches$
```

As an example, consider the following regular expression $R$ and input string $w$:

$$R = (aa + aaa)(aaa + aa)$$
$$w = aaaaabaaaaa$$

We can separate $R$ into two matching groups:

- The first group $(aa + aaa)$ will match either two or three $a$ symbols (e.g. $aaa$ will yield three overlapped matches: $aaa$, $aaa$ and $aaa$).
- The second group $(aaa + aa)$ will also match either two or three $a$ symbols, much like the first group.

The Glushkov automaton construction for $R$ is as follows:

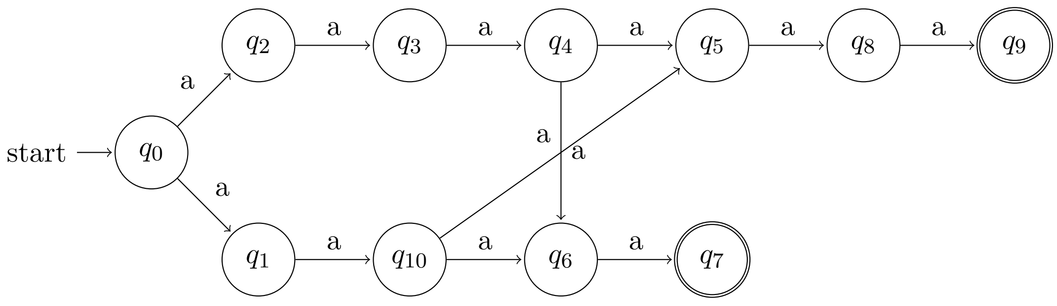

Figure 5.1: Default Glushkov automaton

Meanwhile, the modified position automaton for this regular expression can be constructed using the algorithm presented in 1, resulting in the following:

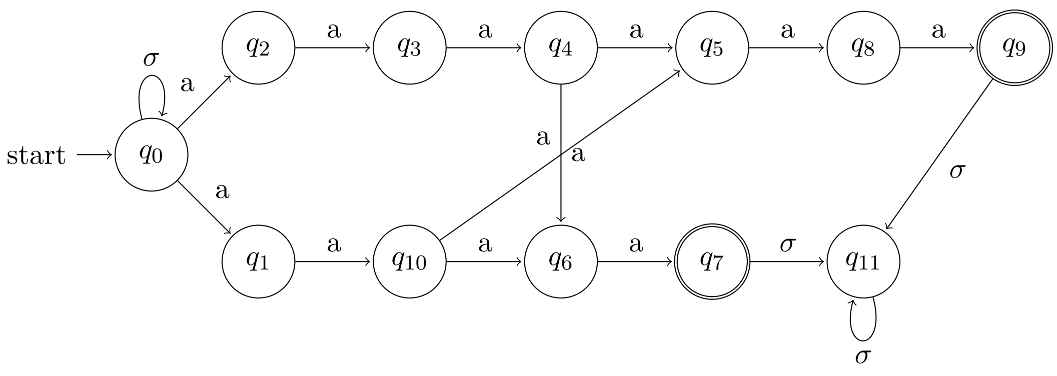

Figure 5.2: Modified Glushkov automaton

When we apply the matching algorithm (2) to an input string, it will traverse the automaton and record all positions where matches occur. This approach ensures that we can find all possible matches, including those that overlap, without falling into the exponential blowup trap of backtracking.

The result is a table that maps each accepting state to a set of index pairs, each indicating the start and end of a successful match. For instance, applying this process to $R = (aa+aaa)(aaa+aa)$ and $w = aaaaabaaaaa$ will yield the following table:

Table 5.1: Match positions table using the regular expression $R = (aa + aaa)(aaa + aa)$ and input string $w = aaaaabaaaaa$

| | $q_0$ | $q_1$ | $q_2$ | $q_3$ | $q_4$ | $q_5$ | $q_6$ | $q_7$ | $q_8$ | $q_9$ | $q_{10}$ | $q_{11}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\varepsilon$ | (0,0) | | | | | | | | | | | |
| a | (1,1) | (0,1) | (0,1) | | | | | | | | | |
| a | (2,2) | (1,2) | (1,2) | (0,2) | | | | | | | (0,2) | |
| a | (3,3) | (2,3) | (2,3) | (1,3) | (0,3) | (0,3) | (0,3) | | | | (1,3) | |
| a | (4,4) | (3,4) | (3,4) | (2,4) | (1,4) | (0,4)<br>(1,4) | (0,4)<br>(1,4) | (0,4) | (0,4) | | (2,4) | |
| a | (5,5) | (4,5) | (4,5) | (3,5) | (2,5) | (1,5)<br>(2,5) | (1,5)<br>(2,5) | (1,5)<br>(0,5) | (1,5)<br>(0,5) | (0,5) | (3,5) | (0,5) |
| b | (6,6) | | | | | | | | | | | |
| a | (7,7) | (6,7) | (6,7) | | | | | | | | | |
| a | (8,8) | (7,8) | (7,8) | (6,8) | | | | | | | (6,8) | |
| a | (9,9) | (8,9) | (8,9) | (7,9) | (6,9) | (6,9) | (6,9) | | | | (7,9) | |
| a | (10,10) | (9,10) | (9,10) | (8,10) | (7,10) | (6,10)<br>(7,10) | (6,10)<br>(7,10) | (6,10) | (6,10) | | (8,10) | |
| a | (11,11) | (10,11) | (10,11) | (9,11) | (8,11) | (7,11)<br>(8,11) | (7,11)<br>(8,11) | (6,11)<br>(7,11) | (6,11)<br>(7,11) | (6,11) | (9,11) | (6,11) |

First, we always have to account for $\varepsilon$, since there is the possibility of having the empty word and we also want to match against it. After that, every symbol $s \in w$ is processed sequentially.

At each step, the algorithm updates a row that maps the automaton's (represented in 5.2) states to sets of position intervals $(i, j)$, such that the substring $w_{ij}$ corresponds to a valid match, whether it is overlapped or not.

Transitions are computed for each input symbol using the automaton's $\delta$ function. When a final state is reached, the interval is stored as a successful match. Furthermore, during this process, only the last symbol's computed transitions and position intervals are preserved because they always carry over to the current symbol's computation.

The automaton on 5.2 shows that:

$$F = \{q_7, q_9\}$$

For those states, the resulting match table yields the following:

Table 5.2: Highlighted substrings of valid matches for state $q_7$

| | |
|---|---|
| $w_{0,4}$ | **aaaaabaaaaa** |
| $w_{0,5}$ | **aaaaabaaaaa** |
| $w_{1,5}$ | **aaaaabaaaaa** |
| $w_{6,10}$ | **aaaaabaaaaa** |
| $w_{6,11}$ | **aaaaabaaaaa** |
| $w_{7,11}$ | **aaaaabaaaaa** |

Table 5.3: Highlighted substrings of valid matches for state $q_9$

| | |
|---|---|
| $w_{0,5}$ | **aaaaabaaaaa** |
| $w_{6,11}$ | **aaaaabaaaaa** |

Recalling the regular expression used $R = (aa + aaa)(aaa + aa)$ and input string $w = aaaaabaaaaa$,

# Bibliography

[1] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996. ISSN: 0304-3975. doi:https://doi.org/10.1016/0304-3975(95)00182-4.

[2] S Broda, M Holzer, E Maia, N Moreira, and R Reis. A mesh of automata. *Information and Computation*, 265:94–111, 2019. doi:10.1016/j.ic.2019.01.003.

[3] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN: 0004-5411. doi:10.1145/321239.321249.

[4] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. Regex and extended regex. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata*, pages 77–84, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN: 978-3-540-44977-5.

[5] Russ Cox. Regular expression matching in the wild, 2010.

[6] Free Software Foundation. grep, .

[7] Free Software Foundation. sed, .

[8] *GNU Awk User's Guide*. Free Software Foundation, 2024.

[9] Google. Re2, a regular expression library.

[10] *The Open Group Base Specifications Issue 8*. The IEEE and The Open Group, 2024.

[11] University of Cambridge. pcre2.

[12] Rogério Reis and Nelma Moreira. Fado: tools for finite automata and regular expressions manipulation. 11 2002.

[13] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN: 0001-0782. doi:10.1145/363347.363387.

[14] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. Re#: High performance derivative-based regex matching with intersection, complement, and restricted lookarounds. *Proceedings of the ACM on Programming Languages*, 9(POPL):1–32, January 2025. ISSN: 2475-1421. doi:10.1145/3704837.

[15] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, Boston, MA, February 2019. USENIX Association. ISBN: 978-1-931971-49-2.

[16] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, Boston, MA, February 2019. USENIX Association. ISBN: 978-1-931971-49-2.

[17] Isaac Z. Schlueter. minimatch, 2011.