Informatics

Inference of Regular Expressions for Ad Hoc Java Parsers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Simon Hofbauer, BSc
Matrikelnummer 11701818

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Mitwirkung: Projektass. Dipl.-Ing. Michael Schröder, BSc

Wien, 27. Jänner 2025

Simon Hofbauer Jürgen Cito

Informatics

Inference of Regular Expressions for Ad Hoc Java Parsers

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Simon Hofbauer, BSc
Registration Number 11701818

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Assistance: Projektass. Dipl.-Ing. Michael Schröder, BSc

Vienna, January 27, 2025

Simon Hofbauer Jürgen Cito

Erklärung zur Verfassung der Arbeit

Simon Hofbauer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. Jänner 2025

Simon Hofbauer

Danksagung

Diese Arbeit stellt das letzte Teilstück auf dem Web zu meinem Masterabschluss dar und möchte ich mich bei all denjenigen bedanken, die mich auf dem Weg zu diesem Abschluss begleitet und unterstützt haben.

Mein besonderer Dank gilt Prof. Jürgen Cito, der nicht nur die Inspiration zu diesem Thema geliefert hat, sondern mich auch während der Durchführung dieser Arbeit als Betreuer unterstützt hat. Trotz seines Forschungsaufenthaltes in den USA war er für mich eine wertvolle Unterstützung, und ich schätze seine Rückmeldungen und Anregungen sehr.

Ebenso möchte ich mich bei meinem Betreuer Dipl.-Ing. Michael Schröder bedanken. Seine Betreuung während der Abwesenheit von Prof. Cito und fachliche Unterstützung waren von unschätzbarem Wert für den erfolgreichen Abschluss dieser Arbeit.

Meinen Eltern möchte ich dafür danken, dass sie mir mein Studium ermöglicht und mir stets den Rücken freigehalten haben. Ohne ihre bedingungslose Unterstützung wäre dieser Erfolg nicht möglich gewesen.

Schließlich gilt mein Dank meinen Freund*innen, Kommilitonen und all denjenigen, die mir während dieser Zeit mit moralischem Beistand, Motivation und Verständnis zur Seite standen.

Acknowledgements

This thesis represents the final step on my journey toward earning my Master's degree, and I would like to express my gratitude to everyone who has supported and accompanied me along the way.

My special thanks go to Prof. Jürgen Cito, who not only provided the inspiration for this thesis but also supported me as my advisor throughout the process. Despite his research stay in the United States, he was an invaluable source of guidance, and I greatly appreciate his feedback and insights.

I would also like to thank my advisor, Dipl.-Ing. Michael Schröder. His supervision during Prof. Cito's absence and his professional support were invaluable for the successful completion of this thesis.

I am deeply grateful to my parents for making my studies possible and for always having my back. Without their unconditional support this achievement would not have been possible.

Finally, I would like to thank my friends, fellow students, and everyone who supported me during this time with moral encouragement, motivation, and understanding.

Kurzfassung

Diese Arbeit befasst sich mit der Ableitung regulärer Ausdrücke (RegEx) aus Ad-hoc-Java-Parsern mithilfe statischer Analyse. Ad-hoc-Parser sind direkt im Quellcode implementiert, verwenden Java-String-Operationen und besitzen meist keine formale Grammatikdefinition. Der Ansatz dieser Arbeit besteht darin, die Weakest Liberal Precondition (WLP) aus der Logik der String-basierten Parser abzuleiten und daraus entsprechende reguläre Ausdrücke zu generieren. Diese RegExes können für das White-Box-Testing genutzt werden, um sowohl gültige als auch ungültige Eingabestrings zu erzeugen. Der methodische Ansatz umfasst:

1. Extraktion des Source Codes und Generierung des Abstract Syntax Tree (AST) mithilfe der Spoon-Bibliothek
2. Berechnung und Vereinfachung der WLP für logikbasierte String-Operationen
3. Ableitung und Transformation regulärer Ausdrücke für Java-String-Methoden sowie Validierung der Ergebnisse

Die Evaluierung erfolgt durch automatisierte Generierung von Java-Parsern, systematische Tests mit RegEx-basierten Eingaben sowie die Anwendung auf praxisnahe Beispiele wie E-Mail-Adress-Parser und Unified-Format-Parser. Die Ergebnisse zeigen die Machbarkeit des Ansatzes bei der Verwendung nur einer String-Methode ohne vorheriger Transformation des Strings, verdeutlichen jedoch Herausforderungen bei der Kombination mehrerer String-Methoden.

Abstract

This thesis focuses on extracting regular expressions (RegEx) from ad hoc Java parsers using static reasoning techniques. Ad hoc parsers often lack formal grammar definitions and are directly implemented in source code using Java string operations. The approach involves deriving the weakest liberal precondition (WLP) of string-based parsing logic to generate corresponding regular expressions. These regular expressions can serve as a model for testing, enabling white-box testing strategies by generating both valid and invalid input strings.
The thesis introduces a systematic methodology:

1. Parsing the Java source code and extracting its Abstract Syntax Tree (AST) using the Spoon library
2. Simplifying code and deriving the WLP for logic-based string functions
3. Generating RegEx representations for Java string methods, including transformations and validations

The evaluation is performed through automated generation of Java parsers, systematic testing with RegEx-derived strings, and real-world use cases like email address and unified format parsers. The results demonstrate the feasibility of the approach for single methods without previous transformation of the string but reveal challenges in concatenated string method calls.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

2 Related Work **3**

3 Background **7**
- 3.1 Regular Expressions 7
- 3.2 Weakest Liberal Precondition 9

4 Approach **11**
- 4.1 Scope 12
- 4.2 Deriving the Abstract Syntax Tree (AST) 14
- 4.3 Extract Weakest Liberal Precondition and Simplify Code 14
- 4.4 RegEx Generation 17

5 Evaluation Methodology **23**
- 5.1 Parsers 24
- 5.2 Generating Strings 25
- 5.3 Metrics 26

6 Results and Discussion **29**
- 6.1 Data set 29
- 6.2 Test Results 30
- 6.3 Discussion 30

7 Conclusion **33**

A Appendix **35**
- A.1 Tutorial on Formal Language Theory 35
- A.2 Hoare Logic 44

A.3 Logic Method Mappers 46
A.4 Transformer mappers 57
A.5 Results 58

Overview of Generative AI Tools Used **65**

List of Figures **67**

List of Tables **69**

List of Algorithms **71**

Bibliography **73**

CHAPTER 1

Introduction

Software testing is a crucial part of modern software engineering. The longer the projects are in development, the more complex the software and the bigger the codebase becomes. This results in the need for automatic software tests and regression tests. By using these tests, one can determine whether a new bug has been introduced in the source code or the software is still working according to its specification.

Some functionality is easier to test compared to others. For example, it is pretty easy to test if a function returns a specific value but when it comes to string testing, it gets more complicated. One often implements parsers as source code, to test if the strings match a certain format. This could be an email address, a telephone number or an International Bank Account Number (IBAN). A parser for an email address would possibly search for the '@' character, and the dot character ('.') followed by a top-level domain ('com', 'de', 'at', and so on). Only if the string matches these requirements, the parser accepts the string as a valid email address and returns the boolean value 'true' (verum). The example in listing 1.1 shows a parser which checks an URL regarding the protocol and returns *true* if the protocol in the URL is "http". The expected resulting regular expression (RegEx) would be `http://.*`.

```java
public class URLParser {
  public boolean parse(String text) {
    String protocol = text.split("://")[0];
    return protocol.equals("http");
  }
}
```

Listing 1.1: Parser example

To test these parsers, one would usually use a kind of random testing strategy, as the range of character strings is countably infinite. But this is pretty inaccurate since one wants to test positive (where the parser returns 'true') and negative (where the parser returns 'false') instances and additionally one would need an external oracle to verify the outcome. To improve testing of ad-hoc parsers, we propose an approach that mines grammars from the source code of the static Java code. Our prototype implementation returns a regular expression which can be used to generate positive and negative instances and to test the parser against them. This transforms the test strategy from a black-box testing approach to white-box testing, and gives us the possibility to perform model based testing on the source code, since regular expressions can be transformed to deterministic finite automata. This gives developers the ability to test the parser against its inferred specification. Additionally, this provides the opportunity for security researchers to try to penetrate the implemented parsers by finding a string for which the parser returns the value ‘true’ even though it should not be accepted, and thus find vulnerabilities.

This work focuses on ad hoc parsers which are implemented in Java, and the corresponding string functions provided by the JDK. Ad hoc parsers are small and simple, intermixed with logic and lack any formal specifications of their input languages [SGC23].

This theses first introduces the related work and what research has already been done in this field, before we take a look at the fundamentals which we need before going into the details of this approach. At the end, we are going to take a look at the results, and figure out the advantages and the problems of this approach. The following research questions are the basis of this work:

RQ1 How can we derive string preconditions for ad hoc Java parsers?
By analysis of the source code and the use of custom methods, we intend to derive the precondition from the parser to transform the precondition into a valid regular expression.

RQ2 Are these derived string preconditions sufficient to capture the full behavior of ad hoc Java parsers?
We intend to use an explorative approach to test if the derived preconditions are sufficient or not.

RQ3 How can we test if the grammar fits the source code?
To test the soundness of the derived grammar, we intend to use the mined grammar to classify strings and compare the outcome to the parser’s result.

CHAPTER 2

Related Work

In this section, we provide an overview over related previously published research in the field of grammar inference as well as grammar testing.

Gopinath et al. [GMZ20] introduce a new algorithm designed to automatically infer readable context-free grammars from a program's input language by examining dynamic control flow. This approach is important for fields like software testing, vulnerability analysis, and reverse engineering, where accurate input specifications are often lacking or outdated. The authors present a method that analyzes the dynamic control flow and character access patterns within recursive descent parsers to generate a context-free grammar without relying on specific program heuristics. Their approach does not depend on data flow to variables and is effective even with advanced parsing methods like parser combinators. Their prototype, named Mimid, was tested on various languages and structures, including JSON, JavaScript, and TinyC, and showed high accuracy in producing readable and precise grammars. This approach overcomes limitations of previous methods, such as dependency on data flows to unique variables, and addresses the challenges of conventional parsing models that rely on specialized patterns or assumptions. The paper demonstrates that the proposed method is both effective and flexible, improving the precision and recall of grammar inference compared to prior methods.

Schröder [Sch22] proposes a grammar inference system which allows programmers to infer input grammars from unannotated source code. Ad hoc parsers are prevalent in code that processes text, using basic string functions to parse input. However, these parsers lack explicit grammar definitions, leading to hidden parsing errors and uncertainties in input handling. Schröder proposes an inference system to automatically deduce input grammars from such parsers, making them safer and more reliable. The core of this approach is Panini, a specialized intermediate language designed to transform ad hoc parsers into a format that allows type and grammar inference. Panini is based on a refined lambda calculus and is not meant for direct execution but for type synthesis. It uses refinement types to model parser behavior accurately.

The inference mechanism identifies the most specific grammar that matches the parser's expected input structure. It utilizes domain-specific knowledge about common parsing patterns, iteratively refining type constraints to generate a complete grammar.

Höschele et al. [HZ16] present a method that automatically derives context-free grammars for program inputs by observing how inputs are processed during execution. Using dynamic taint analysis, the approach tracks input fragments as they propagate through the program and aggregates those handled by the same functions into syntactical entities. The resulting grammar reflects the input structure and can aid in reverse engineering, test input generation, and parsing tasks.

Lin et al. [LZ08] introduce dynamic analysis techniques to automatically derive input syntactic structures for both top-down and bottom-up grammars without requiring source code. For top-down grammars, it analyzes dynamic control dependence to reconstruct the input hierarchy, while for bottom-up grammars, it identifies reductions through stack operations during execution. The proposed methods were tested on real-world applications and demonstrated high accuracy in generating precise input syntax trees.

Blazytko et al. [BAS$^+$19] introduce GRIMOIRE, a fully automated coverage-guided fuzzer designed to efficiently test programs requiring highly structured inputs without human interaction or predefined grammars. Unlike traditional fuzzers, GRIMOIRE synthesizes new inputs dynamically by generalizing and mutating input fragments, resembling grammar-based mutations while avoiding reliance on explicit specifications. The authors demonstrate that GRIMOIRE outperforms state-of-the-art coverage-guided fuzzers and grammar-based approaches, achieving higher code coverage and discovering 19 unique memory corruption bugs, including 11 CVEs.

Caballero et al. [CYLS07] propose an approach called shadowing which extracts protocol message formats by dynamically analyzing program binaries that implement the protocols. Unlike network-trace-based methods, shadowing examines how programs process incoming data, which provides both syntactic and semantic insights. The system, named Polyglot, tracks data flows using dynamic taint analysis to identify field boundaries, direction fields, separators, and protocol keywords. It can detect fixed-length and variable-length fields without assuming predefined encodings or separator values, making it robust for unknown protocols.

Aschermann et al. [AFH$^+$19] present NAUTILUS, a novel fuzzing tool that efficiently finds bugs in programs requiring highly-structured inputs by combining grammar-based input generation with feedback-driven fuzzing. Traditional fuzzers struggle with such programs because their random inputs fail initial syntactic and semantic checks. NAUTILUS overcomes this by leveraging grammars to generate syntactically correct inputs and applying feedback mechanisms like code coverage to guide the fuzzing process, maximizing efficiency. The tool recombines previously discovered valid inputs to explore deeper program logic and identify hidden vulnerabilities. In experiments on targets like ChakraCore, PHP, mruby, and Lua, NAUTILUS significantly outperformed state-of-the-art fuzzers, achieving higher code coverage and discovering multiple critical bugs, resulting in CVEs and monetary rewards. This approach highlights the synergy of grammars and feedback mechanisms, enabling fuzzers to uncover deep, complex program errors effectively.

The thematically most similar publication to this thesis is the work by Schröder [Sch22]. While Schröder transforms the parsers in PANINI (an intermediate language), we try to directly mine the regular expression of each of the string methods applied to the parser's input and to recursively transform the generated RegEx. Schröder evaluated his approach on Python parsers, while we focus on Java parsers. Most of the other research focuses on different structures, like the JSON format or internet protocols. They do not make use of the static source code, but rather of dynamic flows. Gopinath et al. [GMZ20] make use of the dynamic control flow, Höschele et al. [HZ16] use dynamic tainting to trace the data flow of each input character, Lin et al. [LZ08] use dynamic analysis techniques (dynamic control dependence and reductions during execution), Blazytko et al. [BAS+19] try to discover the grammar during fuzzing and Caballero et al. [CYLS07] use dynamic binary analysis.

CHAPTER 3
Background

This chapter gives a short introduction into regular expressions as well as to the Weakest Liberal Precondition, both of them being the formal background of the proposed approach and the prototype implementation. For a short theoretical tutorial on formal language theory in general as well as the Hoare Logic, take a look at the sections A.1 and A.2 in the Appendix.

3.1 Regular Expressions

In this work, we will make use of the regular expression constructs defined by Java in the JDK 23.[1] RegEx provides support for single chars, unicode values as hexadecimals, and custom, predefined, or POSIX character classes (for US-ASCII only), as well as quantifiers and logic operators. The biggest difference between the common RegEx notation and the Java dialect are the special constructs introduced in the JDK which extend the standard notation set. These constructs allow us to make the RegEx much more readable as well as shorter. The two used extensions are positive and negative lookaheads (together also called lookarounds). Positive lookaheads (denoted by `?=`) allow us to check if the remaining characters of the string match the given positive lookahead RegEx, but without moving the cursor. This means, that we can check the remaining characters in the string multiple times. The negative lookahead (denoted by `?!`) allows us to analyze the remaining characters of the current string to not match a certain RegEx, also without moving the cursor. Furthermore we can use the negative lookahead construct to express that there is no string which would match the necessary constraints, like a string with a length < 0.

Even though lookaheads provide the power to change the type of the grammar (from regular to a lower type of grammar, see definition 9), we can be sure to still remain in the

[1]https://download.java.net/java/early_access/valhalla/docs/api/java.base/java/util/regex/Pattern.html

domain of regular languages since we don't use back-references anywhere which would be necessary to violate the constraints for regular languages [CT22, BvdMvL21].

Furthermore, Java offers 3 different subtypes of quantifiers — they can either be greedy, reluctant or possessive [Ora]. Greedy quantifiers try to match as many characters as possible on the first try. If the RegEx does not match the string they start backtracking and reduce the consumed string. Reluctant quantifiers do it the other way around — they try to consume the minimum amount of characters and if the RegEx does not match the given string and they have to backtrack, they increase the amount of consumed characters. Last but not least Java offers possessive quantifiers. These subtypes of quantifiers try to consume the entire string, and only try once to match, meaning that there is no backtracking. This leads to non-matching RegExes, even though they would match if the quantifier would consume less characters.

Table 3.1 gives a short overview over the used expressions for the RegExes in this work.

example	**explanation**
`x`	matches the character x
`\x{h}`	character with hexadecimal code point value 0xh
`abc`	matches the string "abc"
`[abc]`	matches the character a, b or c
`[^abc]`	matches every character but a, b or c
`[a-z]`	matches a character in the range from a to z
`^`	matches the beginning of a string
`$`	matches the end of a string
`X*`	matches X zero or more times
`X{n}`	matches X exactly n times
`X{n,}`	matches X at least n times
`X{n, m}`	matches X at least n but at most m times
`X | Y`	matches either X or Y
`XY`	matches X followed by Y
`(X)`	denotes X as capture group
`a(?=[abc])[a-z]d`	matches the character a, followed by either the character a, b or c, followed by the character d
`a(?!b)[a-c]d`	matches the character a, followed by either the character a or c, followed by the character d

Table 3.1: Examples of RegExes

3.2 Weakest Liberal Precondition

To reverse engineer code and extract the necessary conditions for a string to make the parser evaluate to $True$, the Weakest Liberal Precondition (WLP) is used. This enables the generation of the minimal required precondition (or the largest set of input states), given a postcondition and a statement so that the nondeterministic machine would return in the postcondition state or not at all [Er88]. In other words, the generated precondition is the least restrictive condition needed, that the postcondition holds for the given program. The weakest liberal precondition is denoted by $wlp(p, B)$, where p is the program and B is the postcondition. In contrast to the weakest precondition (denoted as $wp(pB)$), the weakest liberal precondition does not guarantee that the machine will ever reach that state, since it might not terminate.
The following definition can be derived from the Hoare Rules [Hoa69a]:

Definition 1. ***Definition of the Weakest Liberal Precondition***
$wlp(x := a, B) = B[x/a]$
$wlp(skip, B) = B$
$wlp(p_1; p_2, B) = wlp(p_1, wlp(p_2, B))$
$wlp(\mathit{if}\ b\ \mathit{then}\ p_1\ \mathit{else}\ p_2\ \mathit{end}, B) = ((b \implies wlp(p_1, B)) \land (\neg b \implies wlp(p_2, B)))$

We are not looking into the weakest liberal precondition for while loops, since finding inductive invariants is an undecidable problem [KSC+23].

CHAPTER 4
Approach

Java parsers are typically based on Java string methods. Therefore, this work focuses on an approach to generate RegExes for the individual Java string methods and their combination.

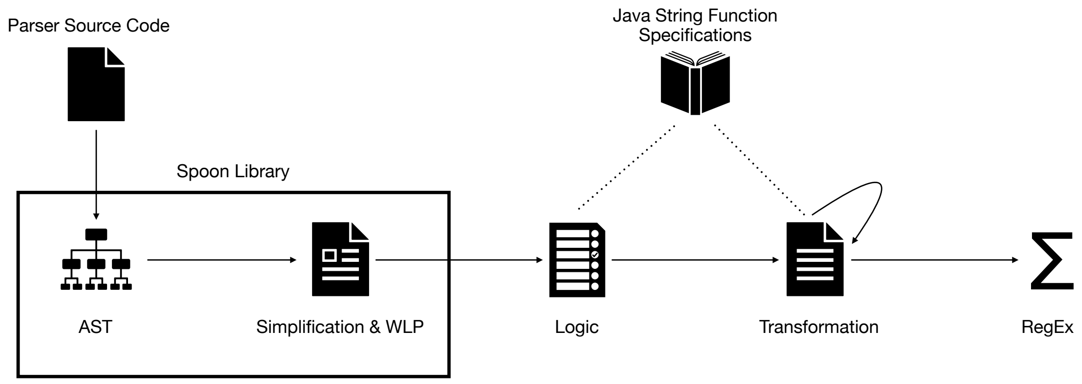


Figure 4.1: Overview of our approach

Algorithm 4.1: Overview of our approach

```
   Input: parser source code
   Output: RegEx
 1 $ast \leftarrow$ transform source code to AST
 2 $wlp \leftarrow$ extract WLP from $ast$
 3 $wlp \leftarrow$ simplify WLP
 4 for $boolExpr\ in\ wlp$ do
 5     $regex \leftarrow$ mine RegEx from $boolExpression$ of logic method
 6     while $transformation\ method\ before\ logic\ method$ do
 7         $regex \leftarrow$ transform $regex$
 8     end
 9     $finalRegex \leftarrow$ append $regex$
10 end
11 return $finalRegex$
```

To work on the parsers' source code, we used the Spoon library [PMP+15], forked the project and adapted the library's code. We implemented 2 major functionalities: (i) extracting the weakest liberal precondition and (ii) mapping it to a RegEx. In the following, we are taking a closer look on each of the different steps, as shown in figure 4.1.

4.1 Scope

We have to limit our approach to parsers from which we can infer the grammar.

4.1.1 Datatypes

We only consider the following datatypes: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double` and `String`. Object types are not within the scope of this work. The *partiallyEvaluate()* method provided by the Spoon library (see section 4.2) offers the functionality to simplify expressions by evaluating and removing those parts, which are not based on the input variable and can therefore be statically evaluated. As it works only on Java primitive types, object types cannot be handled and are therefore excluded.

4.1.2 Structure

All parsers have to return a boolean, which states whether the input is accepted or not.

We only consider parser methods, which have exactly one input parameter of the type `String`. If we would allow more input parameters, we could not generate the grammar statically. An example of a non-inferrable parser is given in listing 4.1. It shows, that we don't know the value of the character which has to be at the given index statically. This information is only available at Runtime. Furthermore the method is only allowed to access variables which are defined in the method and must not access instance or class variables, since we have no information about the value of those variables. The example

```java
public class MultipleInputParametersParser {
  public boolean parse(String text, char value) {
    return text.charAt(4) == value;
  }
}
```

Listing 4.1: Parser example with multiple input parameters

illustrated in listing 4.2 shows that it is not possible to generate the grammar based on the source code including instance variables, since the index could be any integer and cannot be determined statically.

```java
public class NonStaticParser implements Parser {
  public int index;
  @Override
  public boolean parse(String text) {
    return text.charAt(index) == 'a';
  }
}
```

Listing 4.2: Parser example accessing an instance variable

The parsers have to be a straight-line program, meaning that they must not contain any loops since finding the WLP for loops is an undecidable problem. The parsers could contain if-statements, but the condition would have to be expressed with primitives that can be evaluated statically. Therefore, the control flow is already defined when programming the parser as it can be seen in listing 4.3. As this does not offer any additional functionality, we generally do not support if-statements including switch statements or other derived constructs.

```java
public class IfElseParser implements Parser {
  @Override
  public boolean parse(String text) {
    int x = 0;
    boolean result;
    if (x == 0) {
      result = text.charAt(1) == 'a';
    } else {
      result = text.charAt(2) == 'b';
    }
    return result;
  }
}
```

Listing 4.3: Parser containing If-statement with statically evaluate-able condition

Furthermore the parser must contain exactly one return statement. Since we don't support branching in general, there can only be one return statement in a well-formatted code. We assume that the return statement is the last statement in the method, as all statements after the return statement would be ignored anyway when the parser is called. To find the WLP, we have to search for the return statement and replace the intermediate variables in the expression.

Figure 4.2 describes the syntax of the parsers in EBNF. This grammar is a subset of the Java Language Specification [Java].

4.1.3 Characters

We only consider characters in the standard range of chars (also called Basic Multilingual Plane), meaning we do not consider Surrogates. Surrogates are supplemental characters and are represented as a pair of char values. The char data type in Java is based on the original Unicode specification, which encodes characters using 16 bits, allowing values in the range of $0x0000 - 0xFFFF$. Over the time, the Unicode Standard has been extended which created the need for more than 2^{16} characters (the current maximal unicode value is $0x10FFFF$). We only consider Surrogates if it is at the core of the string method, as in the *offsetByCodePoints()* method.

We have to apply this restriction because supporting surrogates would increase the development effort and complexity of the prototype since one character would be represented by two different char instances. In this thesis we want to show the feasibility of this approach and therefore do not support Surrogates.

4.1.4 Method calls

The only method calls we allow are the methods listed in section 4.4. We selected these methods based on the previously introduced restrictions on data types. We only support Java string methods which return a value without implicitly changing the state of the variable, as otherwise our approach of extracting the WLP would no longer work (see section 4.3).

4.2 Deriving the Abstract Syntax Tree (AST)

To access and work directly on the source code we need the Abstract Syntax Tree (AST). The AST gives us a structured access to the source code. The AST is later used to implement and derive the WLP from the parser's source code.

To access the AST and to implement our approach we use the Spoon library (Spoon - Source Code Analysis and Transformation for Java) [PMP+15]. Spoon enables us to analyze, rewrite, transform and transpile Java source code. The library gives us the opportunity to directly access the parts of the different code elements. The AST derived from the parser's source code is used in the next step.

4.3 Extract Weakest Liberal Precondition and Simplify Code

The extraction of the WLP makes intensively use of Spoon Code Elements. Thus, the WLP itself is also represented by Spoon Code Elements. The starting point for extracting the relevant parts of the Java source code is the return statement. It contains all relevant information to find all further relevant code parts.

```
⟨Parser⟩                         ::=‘bool’⟨Identifier⟩‘(String’⟨Identifier⟩
                                     ‘){’[⟨Statements⟩]⟨ReturnStmt⟩‘}’
⟨Statements⟩                     ::=⟨Statement⟩‘;’ | ⟨Statement⟩‘;’⟨Statements⟩
⟨ReturnStmt⟩                     ::=‘return ’⟨Expression⟩
⟨Statement⟩                      ::={⟨Statements⟩} | ⟨VariableDeclarationStatement⟩
                                     | ⟨Expression⟩
⟨VariableDeclarationStatement⟩   ::=⟨Type⟩⟨VariableDeclarators⟩
⟨Expression⟩                     ::=⟨Expression1⟩[‘=’⟨Expression1⟩]
⟨Expression1⟩                    ::=⟨Expression2⟩[⟨Expression1Rest⟩]
⟨Expression1Rest⟩                ::={⟨InfixOp⟩⟨Expression2⟩}
⟨Expression2⟩                    ::=⟨PrefixOp⟩⟨Expression2⟩ | ⟨Primary⟩{⟨Selector⟩}
⟨Selector⟩                       ::=‘.’⟨Identifier⟩[⟨Arguments⟩] | [⟨Expression⟩]
⟨InfixOp⟩                        ::=‘&&’ | ‘||’ | ‘==’ | ‘!=’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘+’
                                     | ‘-’ | ‘*’ | ‘/’
⟨PrefixOp⟩                       ::=‘!’ | ‘+’ | ‘-’
⟨Primary⟩                        ::=⟨Literal⟩ | ⟨ParExpression⟩
                                     | ⟨Identifier⟩{‘.’⟨Identifier⟩}[⟨IdentifierSuffix⟩]
⟨IdentifierSuffix⟩               ::=⟨Expression⟩ | ⟨Arguments⟩
⟨Arguments⟩                      ::=‘(’[⟨Expression⟩{‘,’⟨Expression⟩}]‘)’
⟨Literal⟩                        ::=⟨IntegerLit⟩ | ⟨FloatLit⟩ | ⟨CharLit⟩ | ⟨StringLit⟩
                                     | ⟨BoolLit⟩ | ⟨StringLit⟩
⟨ParExpression⟩                  ::=‘(’⟨Expression⟩‘)’
⟨VariableDeclarators⟩            ::=⟨VariableDeclarator⟩{‘,’⟨VariableDeclarator⟩}
⟨VariableDeclarator⟩             ::=⟨Identifier⟩⟨VariableDeclaratorRest⟩
⟨VariableDeclaratorRest⟩         ::={‘[]’}[‘=’⟨VariableInitializer⟩]
⟨VariableInitializer⟩            ::=⟨ArrayInitializer⟩ | ⟨Expression⟩
⟨ArrayInitializer⟩               ::={‘[’⟨VariableInitializer⟩{‘,’⟨VariableInitializer⟩}‘]’}
⟨Type⟩                           ::=‘byte’ | ‘short’ | ‘char’ | ‘int’ | ‘long’ | ‘float’ | ‘double’
                                     | ‘boolean’ | ‘String’
```

Figure 4.2: Syntax of parsers

Algorithm 4.2: Extracting precondition

```
Input: method as AST
Output: minimized precondition
/* initialize wlp variable                                   */
1 wlp ← find return statement in the AST
/* apply WLP Rule of Composition                             */
2 for stmt in statementsReverseOrder do
3     if stmt is Assignment then
          /* apply WLP Rule of Assignment                    */
4         wlp ← replace assigned variable in wlp by assignment
5     end
6 end
7 minPre ← minimize wlp
8 result ← transform minimizedPrecondition (minPre) using De Morgan
9 return result
```

The algorithms perform the following steps:

1. The first step of extracting the WLP is a modification of the return statement by replacing the intermediate results used in the return statement by those statements which deliver these intermediate results. This step is done by iterating over the statements of the source code in reverse order of the control flow, thus the whole method is reduced to a single compressed statement which represents itself the weakest liberal precondition for this parsing method. The WLP is expressed by Spoon code elements.

2. In the second step, we try to perform a possible minimization of the weakest liberal precondition. Therefore, we use the *partiallyEvaluate* method provided by the Spoon library to get rid of irrelevant statements which can be statically evaluated.

3. As a final step, we transform the statement using De Morgan's Law. When mapping individual statements, we cannot handle double-negated statements or groups of conjunctions or disjunctions with negated statements, as we rely on negative lookahead expressions which behave differently from a logical *NOT* operator. Therefore, we move *NOT* operators as close to the individual statement as possible. For numerical comparators, we transform the comparison operator to remove the negation.

This simplified statement is input to the RegEx-generation in the following step as illustrated in listing 4.4.

```
class ExampleParser implements Parser{
    @Override()
    bool parser(String text){
        bool equals = text.equals("abc");
        bool disjunction = !(!text.startsWith("a") || text.endsWith("b"));
        bool addition = (1 + 2 == 3);
        return equals && addition && disjunction;
    }
}
```

Listing 4.4: Parser example for extracting the WLP

Applying the first step, we replace the intermediate variables in the return statement by the corresponding assignments. Therefore, we end up with the following compressed expression:

$$text.equals(\text{"}abc\text{"}) \; \&\& \; (1 + 2 == 3) \; \&\& \; !(!text.startsWith(\text{"}a\text{"}) \;||\; text.endsWith(\text{"}b\text{"})) \tag{4.1}$$

This is already the weakest liberal precondition, even though it is not minimal. Therefore, we apply the second step to remove the unnecessary mathematical expression $(1+2 == 3)$. This expression does not affect the precondition for the parser, since it is always true. We use the Spoon provided *partiallyEvaluate* method which automatically removes statements like these for primitives. We end up the the following expression:

$$text.equals(\text{"}abc\text{"}) \; \&\& \; !(!text.startsWith(\text{"}a\text{"}) \;||\; text.endsWith(\text{"}b\text{"})) \tag{4.2}$$

In the final step, we work on the negation of the disjunction. We use the De Morgan's Law to move the negation into the disjunction, ending up with the following final precondition which is used for the mapping later on:

$$text.equals(\text{"}abc\text{"}) \; \&\& \; (text.startsWith(\text{"}a\text{"}) \; \&\& \; !text.endsWith(\text{"}b\text{"})) \tag{4.3}$$

It is to be noted that the final precondition is still not minimal regarding the logic of the used methods. The redundant conditions cannot be eliminated without analyzing the logic of these methods. This might lead to an unnecessary complex RegEx result. Figure 4.3 describes the syntax of the resulting WLP, with respect to the previously given sytax in section 4.1. It is represented by Spoon code elements.

$$\langle \texttt{WLP} \rangle ::= \langle \texttt{Expression1} \rangle$$

Figure 4.3: Syntax of the WLP

4.4 RegEx Generation

In this section, we illustrate how we can generate a RegEx from the derived precondition.

For each considered Java string method, we developed a mapper which transforms the string method call including the input parameters and the comparison value to the respective RegEx. Using the Java documentation [Javb], every method had to be analyzed in depth to find a procedure for generating the matching RegEx. We developed the mappers in a test-driven development process, i.e. by implementing parsers calling the considered Java string method with typical input parameters if needed, mining the grammar using the mapper under development and testing the resulting grammar. Based on the results of the tests, the mappers have been improved. We have no proof of the correctness of the mappers, since testing can only show the existence of bugs but can never proof correctness. Depending on the complexity of the considered Java string method, these analyses had to take into account all possible combinations of input parameters, including even unsupported sets, which can cause an exception. Therefore, the sizes of the resulting mappers varies strongly, as can also be seen in the mappers presented later on.

We distinguish between logic and transformer methods. Logic methods return a primitive value and can therefore be compared directly. Transformer methods return a string as they transform a string into a different string before it gets fed into a logic method.

4.4.1 Logic Methods

Applying the limitation stated in section 4.1, we only consider the logic methods from the JDK 23 [Javb] listed in table 4.1.

Some of the methods listed in table 4.1 are overloaded (in the context of Java, *overloaded* means that the methods have the same name, but a different amount or type of parameters). In several cases, overloaded methods can be handled by the mapper of the most unspecific method implementation, where additional input parameters can be replaced by the corresponding default values. Therefore, some of the mappers of the methods listed in table 4.1 can be replaced by other mappers. For example the method call $isEmpty() == true$ can be transformed to the method call $length() == 0$ or $isEmpty() == false$ can be transformed to $length() > 0$.

It is obvious, that some of these methods are easier to map to a RegEx, while others are much more complex. For example string methods which return a boolean are easier to map compared to a method which returns an integer, since the return value of a boolean can either be *true* or *false* (only 2 possible values), while an integer can be in the range of -2^{31} to $2^{31} - 1$ and these values might not be mapped in a linear manner (for example, the value -1 often has a special meaning). Thus, to map methods which do not deliver a boolean but any numeric type, we have to create a boundary value analysis for each of these methods for the transformation process.

For the WLP $text.isEmpty() == true$, the mapper generates the RegEx (), which is equivalent to an empty string ε. For the WLP $text.isEmpty() == false$, the mapper returns a RegEx like `(.){1,}` meaning it can be any character and the variable text must have a length of at least 1 character.

method signature	**replacement**
char charAt(int index)	codePointAt(int index) (char can be converted to an integer value)
int codePointAt(int index)	-
int codePointBefore(int index)	codePointAt(int index - 1)
int codePointCount(int beginIndex, int endIndex)	-
int compareTo(String anotherString)	-
int compareToIgnoreCase(String str)	compareTo(String str) in combination with the toUpperCase() and toLowerCase() transformers
boolean contains(CharSequence s)	-
boolean contentEquals(CharSequence s)	equals(Object anObject) (since CharSequence can only be a String object because we don't consider any other objects as defined in section 4.1)
boolean endsWith(String suffix)	-
boolean equals(Object anObject)	compareTo(String anotherString) (map *true* to $= 0$ and *false* to $\neq 0$)
boolean equalsIgnoreCase(String anotherString)	compareToIgnoreCase(String str) (map *true* to $= 0$ and *false* to $\neq 0$)
int indexOf(int ch)	indexOf(int ch, 0)
int indexOf(int ch, int fromIndex)	-
int indexOf(String str)	indexOf(String str, 0)
int indexOf(String str, int fromIndex)	-
boolean isBlank()	-
boolean isEmpty()	length() (map *true* to $= 0$ and *false* to $\neq 0$)
int lastIndexOf(int ch)	-
int lastIndexOf(int ch, int fromIndex)	-
int lastIndexOf(String str)	-
int lastIndexOf(String str, int fromIndex)	-
int length()	-
boolean matches(String regex)	-
int offsetByCodePoints(int index, int codePointOffset)	-
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	-
boolean regionMatches(int toffset, String other, int ooffset, int len)	regionMatches(false, int toffset, String other, int ooffset, int len)
boolean startsWith(String prefix)	startsWith(String prefix, 0)
boolean startsWith(String prefix, int toffset)	-
char[] toCharArray()	-

Table 4.1: Considered Java String Logic Methods

Take a look at a more complex mapper, like the *codePointAt(index)* mapper. The *codePointAt(index)* Java method returns the code point of the string at a given index. In Java, characters can be converted to integer values and therefore numeric comparators can be applied ($=, \neq, <, >, \leq, \geq$). We have to implement 4 different mappers since the remaining two comparators can be transformed by modifying the reference value (the comparator $\leq$ can be transformed to $<$ and $\geq$ can be transformed to $>$). For the development process, we first created a boundary value analysis for each operator and implemented the mappers in a second step. Some examples for the mappers are given in figures 4.5 and 4.6. The remaining mappers can be seen in the appendix in section A.3. Their conditions might not be mutually exclusive, therefore they have to be evaluated from top to bottom, since later conditions might be weaker.

For the sake of visible separation, we have set the color of RegExes to blue and the color of variables to red. For the mappers description we use the notation defined in figure 4.4.

$$
\begin{aligned}
&[\![\cdot]\!]_{\mathbb{Z}} && : \mathsf{Java} \hookrightarrow \text{integer} \\
&[\![\cdot]\!]_{\mathbb{B}} && : \mathsf{Java} \hookrightarrow \text{bool} \\
&[\![\cdot]\!]_{\mathbb{S}} && : \mathsf{Java} \hookrightarrow \text{String} \\
&[\![\cdot]\!]_{\text{RE}} && : \mathsf{Java} \hookrightarrow \text{RegEx} \\
&\sum_{i=0}^{n}(x_i) && \doteq x_0 x_1 \ldots x_n \\
&\Big|_{i=0}^{n}(x_i) && \doteq x_0 | x_1 | \ldots | x_n \\
&\text{str[n]} && \doteq \text{character of the string str at index n} \\
&\text{str[i : n]} && \doteq \text{str[i] str[i + 1]} \ldots \text{str[n]} \\
&regex \xrightarrow{\text{inject}} [\![method]\!]_{\text{RE}} && \doteq \text{injects the given regex to the mapper}
\end{aligned}
$$

Figure 4.4: Definition of Notation

$$[\![t.\texttt{codePointAt}(int\ index) == c]\!]_{\text{RE}} \rightarrow .\{[\![\text{index}]\!]_{\mathbb{Z}}\}\backslash\texttt{x}\{[\![\text{c}]\!]_{\mathbb{Z}}\}.*$$

Figure 4.5: Example of Mapper Notation

For the sake of readability we remove the semantic bracketing of variables from the mappers' definition, resulting in mapper descriptions shown in figure 4.6. All logic mapper definitions can be found in the appendix in section A.3.

4.4.2 Transformation Methods

A transformation method transforms an input string into a different output string. A string can never be the result of a parser since our parsers have to return a boolean value

$$[\![t.\texttt{codePointAt}(int\ index) == c]\!]_{\mathrm{RE}} \rightarrow \texttt{.\{index\}\textbackslash x\{c\}.*}$$
$$[\![t.\texttt{codePointAt}(int\ index) \neq c]\!]_{\mathrm{RE}} \rightarrow \texttt{.\{index\}[\^{}\textbackslash x\{c\}].*}$$
$$[\![t.\texttt{codePointAt}(int\ index) < c]\!]_{\mathrm{RE}} \rightarrow \texttt{.\{index\}[\textbackslash x\{0\}} - \texttt{\textbackslash x\{}c - 1\texttt{\}].*}$$
$$[\![t.\texttt{codePointAt}(int\ index) > c]\!]_{\mathrm{RE}} \rightarrow \texttt{.\{index\}[\textbackslash x\{}c + 1\texttt{\}} - \texttt{\textbackslash x\{ffff\}].*}$$
$$[\![t.\texttt{endsWith}(String\ aStr) == c]\!]_{\mathrm{RE}} \rightarrow \begin{cases} .{*}\mathrm{aStr}\$ & c = true \\ .*\ (?!\mathrm{aStr}) & \text{otherwise} \end{cases}$$
$$[\![t.\texttt{length}() == c]\!]_{\mathrm{RE}} \rightarrow \begin{cases} ?! & c < 0 \\ .\{\mathrm{c}\} & \text{otherwise} \end{cases}$$

Figure 4.6: Definition of some Logic Method Mappers

(see limitations in section 4.1). Thus, there always has to be a logic function at the end of a call chain. Therefore, we first map the logic method using the mappers introduced in section 4.4.1. The resulting RegEx from the logic method is transformed recursively, depending on the transformation methods in the call chain.

Table 4.2 lists the considered transformation methods.

method signature	**replacement**
String concat(String str)	
String repeat(int count)	
String replace(char oldChar, char newChar)	replace(String, String)
String replace(String target, String replace)	
String replaceFirst(String regex, String replace)	
String[] split(String regex)	
String strip()	stripLeading() and stripLeading()
String stripLeading()	
String stripTrailing()	
String substring(int beginIndex)	
String substring(int beginIndex, int endIndex)	
String toLowerCase()	
String toUpperCase()	
String toLowerCase(Locale locale)	
String trim()	

Table 4.2: Considered Java String Transformation Methods

The mappers of some methods are fairly easy to implement, while others are more complicated. For example, for implementing the *trim*() method mapper (first mapper in figure 4.7), we simply have to surround the previously mined RegEx by `[\u0-\u20]*`.

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{trim}()]\!]_{\mathrm{RE}} \rightarrow [\backslash\texttt{x}\{\texttt{0x0}\} - \backslash\texttt{x}\{\texttt{20}\}]*\text{regex}[\backslash\texttt{x}\{\texttt{0x0}\} - \backslash\texttt{x}\{\texttt{20}\}]*$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{stripLeading}()]\!]_{\mathrm{RE}} \rightarrow$$
$$((\backslash\texttt{x}\{\texttt{9}\})|(\backslash\texttt{x}\{\texttt{a}\})|(\backslash\texttt{x}\{\texttt{b}\})|(\backslash\texttt{x}\{\texttt{c}\})|(\backslash\texttt{x}\{\texttt{d}\})|(\backslash\texttt{x}\{\texttt{1c}\})|(\backslash\texttt{x}\{\texttt{1d}\})|(\backslash\texttt{x}\{\texttt{1e}\})|(\backslash\texttt{x}\{\texttt{1f}\})|(\backslash\texttt{x}\{\texttt{20}\})$$
$$|(\backslash\texttt{x}\{\texttt{1680}\})|(\backslash\texttt{x}\{\texttt{2000}\})|(\backslash\texttt{x}\{\texttt{2001}\})|(\backslash\texttt{x}\{\texttt{2002}\})|(\backslash\texttt{x}\{\texttt{2003}\})|(\backslash\texttt{x}\{\texttt{2004}\})|(\backslash\texttt{x}\{\texttt{2005}\})$$
$$|(\backslash\texttt{x}\{\texttt{2006}\})|(\backslash\texttt{x}\{\texttt{2008}\})|(\backslash\texttt{x}\{\texttt{2009}\})|(\backslash\texttt{x}\{\texttt{200a}\})|(\backslash\texttt{x}\{\texttt{2028}\})|(\backslash\texttt{x}\{\texttt{2029}\})|(\backslash\texttt{x}\{\texttt{205f}\})$$
$$|(\backslash\texttt{x}\{\texttt{3000}\}))*\text{regex}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{split}(String\ split)[\texttt{i}]]\!]_{\mathrm{RE}} \rightarrow$$
$$\left(\sum_{n=0}^{\mathrm{i}-1}\left(((?!\text{split}).)*\right)(\text{regex})\right)|\left(\sum_{n=0}^{\mathrm{i}-1}\left(((?!\text{split}).)*\right)(\text{regex})\text{split}.*\right)$$

Figure 4.7: Some Mappers for Transformation Methods

Other transformation method mappers like the *repeat*() mapper are of higher complexity and are only partially considered in this work. Therefore, our approach does not provide full coverage of all possible generated RegExes. To cover the remaining field we would have to implement custom parsers and interpreters (e.g. for the $matches(String\ regex)$ logic method) to transform the given RegEx (which is a string object) into our own RegEx data types. Furthermore, we would have to get rid of the positive and negative lookaheads, which we use for the sake of simplicity and readability of our RegExes. For example, we have to limit the *concat* or the *repeat* method to only consider our implementation of *StringRegex*, and we don't allow any encoded unicode characters, character classes or lookarounds. Such transformations are out of scope of this thesis. Not supported RegEx input to the mappers raise a *NotSupportedException*. Multiple transformations methods can be called sequentially, for example $text.stripLeading().stripTrailing().equals(\text{"}abc\text{"})$. This is supported by our approach. The remaining transformations mappers are listed in section A.4.

CHAPTER 5

Evaluation Methodology

The scope of the evaluation is to test whether our approach can generate correct RegExes for the parsers and to measure whether they over- or underestimate. Our evaluation methodology is described in this chapter. Figure 5.1 gives an overview of our custom evaluation pipeline.

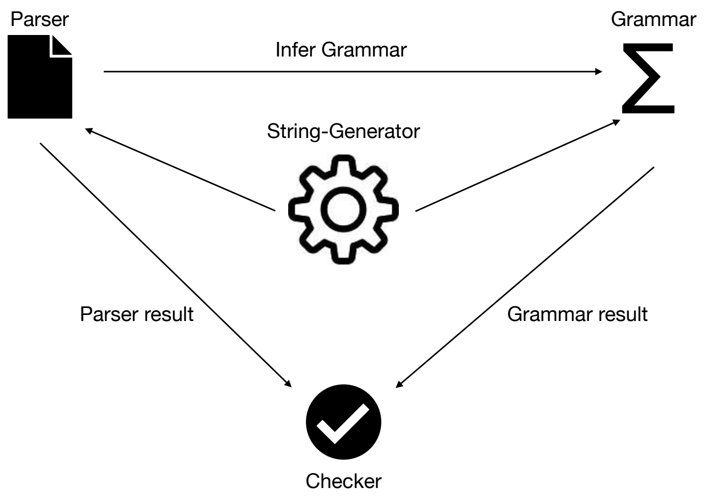


Figure 5.1: Evaluation pipeline

We automatically generate Java parsers by combining the considered methods and different values, apply our approach to infer the grammar and generate strings, classifying them whether they match the extracted regular expression and testing them against the parser.

5.1 Parsers

To test our approach, we need parsers from which we can infer the grammar.

5.1.1 Automatic Parser Generation

The first step of the evaluation pipeline is the generation of parsers which are used to infer their grammars and as oracles to check the correctness of the mined RegExes. These parsers use Java string methods listed in section 4.4.

5.1.2 Real Use Cases

In addition to the automatically generated parsers above, we have implemented 5 more parsers with real use cases. These parsers are the Email-parser (illustrated in listing 5.1), the Unified Format parser (listing 5.2), the HTTP Url parser (listing 5.3), the Time parser (listing 5.4) and the TU Wien Url parser (listing 5.5). An example for a Unified Format would be *@@ -2,8 +3,5 @@* (for further information take a look at GNU Unified Format).[1]

For each of the real use case parsers we manually determined a "golden grammar" to generate matching and non-matching strings, to test the infered grammar against the golden grammar.

```java
public class EmailParser implements Parser {
  @Override
  public boolean parse(String text) {

    String[] atParts = text.split("@");
    boolean firstPart = atParts[0].matches("[a-z0-9.]+");
    String[] dotParts = atParts[1].split("\\.");
    boolean secondPart = dotParts[0].matches("[a-z0-9]+");
    boolean thirdPart = dotParts[1].matches("[a-z]{2,6}");

    return firstPart && secondPart && thirdPart;
  }
}
```

Listing 5.1: Email parser

```java
public class UnifiedFormatParser implements Parser {
  @Override
  public boolean parse(String text) {
    boolean startsWithResult = text.startsWith("@@ ");
    boolean endsWithResult = text.endsWith(" @@");
    String textWithoutLeadingAt = text.substring(3);
    String[] split = textWithoutLeadingAt.split(" ");
    boolean firstSection = split[0].startsWith("-");
    boolean firstSectionComma = split[0].contains(",");
    boolean secondSection = split[1].startsWith("+");
    boolean secondSectionComma = split[1].contains(",");
    return startsWithResult && endsWithResult && firstSection && secondSection && firstSectionComma
      && secondSectionComma;
  }
}
```

Listing 5.2: Unified Format parser

[1]https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html

```java
public class HttpUrlParser implements Parser {
  @Override
  public boolean parse(String text) {
    String protocol = text.split("://")[0];
    return protocol.equals("http");
  }
}
```

Listing 5.3: HTTP Url parser

```java
public class TimeParser implements Parser {
  @Override
  public boolean parse(String text) {
    String regex0To60 = "(([0-9])|(1[0-9])|(2[0-9])|(3[0-9])|(4[0-9])|(5[0-9]))";

    String[] parts = text.split(":");
    boolean hours = parts[0].matches("(([0-9])|(1[0-9])|(2[0-3]))");
    boolean min = parts[1].matches(regex0To60);
    boolean sec = parts[2].matches(regex0To60);

    return hours && min && sec;
  }
}
```

Listing 5.4: Time parser

```java
public class TUWienUrlParser implements Parser {
  @Override
  public boolean parse(String text) {
    String temp = text.strip();
    return temp.startsWith("tuwien",11);
  }
}
```

Listing 5.5: TU Wien Url parser

5.2 Generating Strings

To generate test strings we implemented two approaches with different strategies.

5.2.1 Exhaustive testing up to length 2

We generated all strings with the char-points in the range of $0x10 - 0x7F$ and $length \leq 2$. This results in $1 + (0x7F - 0x10) + (0x7F - 0x10)^2 = 12,433$ input strings. These generated strings are independent of the mined grammar.
Each of the generated strings is classified by the Checker using the extracted RegEx. The result of the classification is checked against the respective parser's returned result. But it is clear that this string generation concept is not sufficient, since certain string methods would never be able to evaluate to either *true* or *false*, depending on other input parameters. A requirement like $length > 10$ could never be fulfilled by any input string of a length limited by 2. The exhaustive testing concept has been chosen to make sure that we test a lot of different inputs without a possible bias of the used library, which is explained in the next section.

5.2.2 Grammar based testing

We use the mined regular expression to generate targeted matching and not matching strings by usage of a library which takes the regular expression as input. These strings are then rechecked if they match or don't match the regular expression, since the used RgxGen[2] library can't handle all of the Java dialect specific RegEx expressions like negative and positive lookaheads. This library applies a state machine to create strings (see section A.1). The returned strings are not limited to length 3 (with respect to the grammar). We try to generate *100,000* matching and not-matching strings each and test them using the parser as an oracle.

During testing we discovered some interesting behavior of the RgxGen String generator. For example, it can happen that the generator does not terminate in a reasonable time or that the library always generates the same string. To relieve this problem, we implemented a timeout of 10 seconds for generating and testing *100,000* positive and negative strings each.

The combination of the exhaustive testing and the grammar testing approach assures to cover at least a subset of all strings with $length < 3$ and additionally a selection of strings without length limitation.

5.3 Metrics

In this section, we describe the metrics which we use to measure the correctness of our mapping. We make use of the well known performance metrics of precision and recall. These metrics are commonly used when it comes to evaluating classification tasks.

5.3.1 Precision

The precision is the fraction of relevant instances among the instances found. It is defined by the following formula:

$$Precision = \frac{\#truePositives}{\#truePositives + \#falsePositives} \tag{5.1}$$

The precision describes the fraction of correctly labeled positive instances over all positive labeled instances. It answers the question: Of the items labeled as positive, how many were correctly identified? The metrics' value range is $[0.00 - 1.00]$. The best achievable value is 1.00, the worst 0.00. A low precision value means, that our approach tends to overestimate.

5.3.2 Recall

The recall describes the fraction of the relevant instances which have been retrieved. It is defined by the following formula:

$$Recall = \frac{\#truePositives}{\#truePositives + \#falseNegatives} \tag{5.2}$$

[2]https://github.com/curious-odd-man/RgxGen

The recall is defined as the fraction of the correctly labeled positive instances over the real positive instances. It answers the question: Of all the actual positive items, how many were correctly identified? The metrics' value ranges from $[0.00 - 1.00]$. The best achievable value is 1.00, the worst 0.00. A low recall value means, that our approach tends to underestimate.

To evaluate the recall, we use a manually written golden grammar to generate *1,000,000* matching and non-matching input strings each and test our inferred grammar against it.

5.3.3 Classification

To calculate the metrics introduced earlier, we have to classify the tested input strings. This is done according to the following scheme:

Class	**Parser result (reference)**	**Grammar result**
truePositive	True	True
falseNegative	True	False
falsePositive	False	True
trueNegative	False	False

Table 5.1: Classification of single input strings

True positives are also referred as "hit", true negatives as "correct rejection", false positives as "overestimation" and false negatives as "underestimation". We can make use this those labels in the discussion to clarify if our approach causes an over- or underestimation.

CHAPTER 6

Results and Discussion

6.1 Data set

A small Python script generated several thousand (not-distinctive) parsers in a systematic way. Each parser uses only one string method as listed in table 4.1. Since many of the listed Java string methods require parameters, we have to generate these values as well. The sets in equation 6.1 show which argument values were used for the specific types.

$$\begin{aligned}
bool &= \{true, false\}\\
String &= \{\text{“}a\text{”}, \text{“}abc\text{”}, \text{“}x\text{”}, \text{“@”}, \text{“”}, \text{“␣”}, \text{“}klmn\text{”}\}\\
char &= \{\text{‘a’}, \text{‘b’}, \text{‘c’}, \text{‘z’}, \text{‘␣’}, \text{‘@’}\}\\
int &= \{-5, -1, 0, 1, 2, 5, 10, 15\}\\
Locale &= \{\text{CANADA}, \text{CANADA-FRENCH}, \text{CHINESE}, \text{ENGLISH},\\
&\quad \text{FRANCE}, \text{FRENCH}, \text{GERMAN}, \text{JAPAN}, \text{ROOT}\}
\end{aligned} \tag{6.1}$$

We generated all possible parsers combining the used string methods of table 4.1 with the type-fitting values of the sets in 6.1 as input parameters. This resulted in 61,692 distinctive parsers which we used to test our approach.
Furthermore, we tried to concatenate the string methods depending on matching datatypes. We generated all tests with less than 3 string method calls. Since the combination of the transformation methods with the logic methods delivers already a high amount of parsers, the combination of these parsers with all given arguments resulted in an infeasible number of parsers, therefore we had to limit the arguments to a single default value for each of the different datatypes. This resulted in an additional 1,635 parsers.

6.2 Test Results

Table 6.1 shows the precision and recall of our approach. Since we cannot create the golden grammar for the multitude of automatically generated parsers by hand, we cannot provide the recall for those instances. The mapping time describes the time the mapping requires, i.e. to map a parser's source code to the corresponding RegEx.

Class	**Number of parsers**	**Precision**	**Recall**	**Mapping Time in seconds**	
				mean	**stdev**
logic only	61,692	1.00	-	6.383	23.866
combined	1,635	0.980	-	0.716	14.590
E-Mail Parser	1	1.00	1.00	< 0.001	-
Unified Format Parser	1	1.00	1.00	< 0.001	-
HTTP Url Parser	1	1.00	1.00	< 0.001	-
Time Parser	1	1.00	1.00	< 0.001	-
TUWien Url Parser	1	>0.999	1.00	< 0.001	-

Table 6.1: Resulting performance metrics

To perform these tests, we used a MacBook Pro 2021 containing an Apple M1 Max Chip with 10 Cores and 64 GB RAM. The failed tests are listed in table A.1 in the Appendix.

6.3 Discussion

Based on the results illustrated in 6.1 we can see that our approach works fine if the parser uses only one string method (logic only). We achieved a precision of 1.0 (100%). But if we concatenate (combine) transformation methods with logic methods, we can see that the precision decreases to 0.98 (98%). Therefore our approach tends to overestimate slightly. Based on the precision and recall we conclude that the mappers developed in our approach are highly accurate.

During testing, we could see that the mapping time varies a lot. It depends on the algorithm used to transform specific Java string method to a RegEx. Especially for mappers which require an extensive enumeration of all possible cases, the mapping time can increase significantly (e.g. exponential growth with the *codePointCounts()* method, see definition of the mapper in the Appendix section A.3).

As we can see in the table A.1 in the Appendix, all failing parsers make use of a transformation method that removes characters from the string, namely the methods *strip()*, *stripLeading()*, *stripTrailing()* and *trim()*. Listing 6.1 illustrates one of the failing examples. This parser returns true, if the text at index 1 is equal to the character 'a' after applying the strip method to the input variable. The strip method removes all the

```java
public class FailingParserExample implements Parser {
  @Override
  public Boolean parse(String text) {
    return text.strip().charAt(1) == 'a';
  }
}
```

Listing 6.1: Failing parser example

leading and trailing white spaces of a string. A white space can for example also be a *carriage return* or a *form feed* character (as defined in the JDK 23 [Javb]). Let's take a look how our approach generates the RegEx. The charAt-logic returns a RegEx like

$$charAtRegex = .\{1\}a(.)* \tag{6.2}$$

while the strip-transformation mapper surrounds the *charAtRegex* with the RegEx:

$$\begin{aligned} stripRegex =& ((\backslash x\{9\}) \mid (\backslash x\{a\}) \mid (\backslash x\{b\}) \mid (\backslash x\{c\}) \mid (\backslash x\{d\}) \\ & \mid (\backslash x\{1c\}) \mid (\backslash x\{1d\}) \mid (\backslash x\{1e\}) \mid (\backslash x\{1f\}) \mid (\backslash x\{20\}) \\ & \mid (\backslash x\{1680\}) \mid (\backslash x\{2000\}) \mid (\backslash x\{2001\}) \mid (\backslash x\{2002\}) \mid (\backslash x\{2003\}) \\ & \mid (\backslash x\{2004\}) \mid (\backslash x\{2005\}) \mid (\backslash x\{2006\}) \mid (\backslash x\{2008\}) \mid (\backslash x\{2009\}) \\ & \mid (\backslash x\{200a\}) \mid (\backslash x\{2028\}) \mid (\backslash x\{2029\}) \mid (\backslash x\{205f\}) \mid (\backslash x\{3000\}))* \end{aligned} \tag{6.3}$$

This results in the concatenation of the RegExes as follows:

$$resultRegex = stripRegex\ charAtRegex\ stripRegex \tag{6.4}$$

Even though all generated RegExes parts are correct by themselves, their combination leads to an imprecise RegEx. The *charAtRegex* allows the first character to be a whitespace, while the strip method removes that whitespace. This leads to a problem. To solve this specific issue we would have to exclude whitespaces from being the first character in the *charAtRegex*. This example illustrates, that the outer RegEx (*stripRegex*) can influence the inner RegEx (*charAtRegex*). Our approach cannot handle such dependencies.

CHAPTER 7

Conclusion

In this thesis, we have shown that it is possible to transform Java string methods to RegExes. It works better for the logic methods, i.e. returning primitives, but becomes inaccurate as soon as we apply transformation methods i.e. returning strings.
In further research one could look at an approach to remove the lookaround expressions and to be able to fix the presented issues when it comes to transformation of RegExes.
To sum up we can answer the research questions as follows:

RQ1 How can we derive string preconditions for ad hoc Java parsers?
We can derive the string preconditions by using the Weakest Liberal Precondition axioms derived by Hoare calculus in combination with the abstract syntax tree. The resulting Weakest Liberal Precondition can be transformed to a regular expression using specific mappers for each Java string method.

RQ2 Are these derived string preconditions sufficient to capture the full behavior of ad hoc Java parsers?
For parsers containing concatenated string methods we could clearly see that this approach is not sufficient. For the logic methods we have no hint that the derived string preconditions are not sufficient, since several thousand parsers making use of different string methods and several thousand strings have been tested on each of the different parsers. However, we have also no formal proof of the sufficiency of the string preconditions.

RQ3 How can we test if the grammar fits the source code?
To test whether the extracted grammar fits the source code, we performed two different string generator concepts. One of these concepts depends on the grammar, while the other is independent. By classifying the generated string and comparing the classification outcome to the parser's result, we tested the soundness of the grammar.

APPENDIX A

Appendix

A.1 Tutorial on Formal Language Theory

In the 1950's, Noam Chomsky attempted to give an accurate description of the structure of natural languages according to mathematical rules. [MAK88] These rules are called grammars (denoted as G_n). These grammars generate all the words in the corresponding language (denoted as L_n).

Definition 2. ***Grammars*** *[MAK88]*
A grammar G is a quadruple (V_T, V_N, S, P) where V_T and V_N are disjoint finite alphabets, and

1. *V_T is the terminal alphabet for the grammar;*
2. *V_N is the nonterminal or variable alphabet for the grammar;*
3. *S is the start symbol for the grammar; and*
4. *P is the grammars set of productions. P is a set of pairs (v, w) with v a string on $V_T \cup V_N$ containing at least one nonterminal symbol, while w is an arbitrary element of $(V_T \cup V_N)*$. An element (v, w) of P is usually written $v \to w$.*

In other sources, the order of the elements in the quadruple might be different (like [Pet22]). In this work, we are referencing the terminal alphabet as V_T, the non-terminal alphabet as V_N, the start symbol as S and the set of productions as P. The variable V is defined as $V_T \cup V_N$. The elements of the terminal alphabets are usually denoted as lower-case Latin letters (a, b, ..., z), while elements of the non-terminal alphabet are usually denoted as upper-case Latin letters (A, B, ..., Z). [Pet22] The empty word is denoted as ε and has a length of 0 (denoted as $\mid \varepsilon \mid = 0$). For the sake of less productions, this work makes use of the group notation, meaning that the set of productions $\{S \to a, S \to b\}$ can be written as $\{S \to a|b\}$. [Pet22]

Every grammar generates a language (set of words), but one language can be generated by several different grammars. [Gyö12]

Definition 3. ***Equivalence***
Two grammars are called equivalent, if they generate the same language.

Let's take a look at some examples.

Example 1. $\{a^n \mid a \geq 0\}$
$G_1 = (\{a\}, \{S\}, S, \{S \to \varepsilon, S \to aS\})$

Example 1 shows the definition of a grammar which can be used to generate all words of the form $\{a^n \mid a \geq 0\}$.

Example 2. $\{a^n b^n \mid a \geq 0\}$
$G_2 = (\{a, b\}, \{S\}, S, \{S \to \varepsilon, S \to aSb\})$

Example 2 shows the definition of the grammar which can be used to generate all words of the form $\{a^n b^n \mid a \geq 0\}$.

Example 3. *Palindromes*
$G_3 = (\{a, b\}, \{S\}, S, \{S \to aSa|bSb|a|b|\varepsilon\})$

Example 3 shows the definition of the grammar which can be used to generate all palindromes over the letters "a" and "b". A palindrome is a word, which is spelled the same from the front and the back (denoted as $w = w^r$).

Example 4. $\{a^n b^n c^n \mid n \geq 0\}$
$G_4 = (\{a, b, c\}, \{S, B, C\}, S, \{S \to aSBC|aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc\})$

We can also perform operations on the languages. Since languages are defined as sets of words, we can perform set-theoretical operations on them. [Pet22]

Definition 4. ***Operations on Languages***

- *Complement:* $\overline{L_1} = \{w \in \overline{L_1} : w \notin L_1\}$
- *Union:* $L_1 \cup L_2 = \{w | w \in L_1 \text{ or } w \in L_2\}$

- *Intersection:* $L_1 \cap L_2 = \{w | w \in L_1 \text{ and } w \in L_2\}$
- *Difference:* $L_1 - L_2 = \{w | w \in L_1 \text{ and } w \notin L_2\}$
- *Concatenation:* $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$
- *Potency:* $L^n = \{w_1 ... w_n \mid w_1, ..., w_n \in L\}$
- *Star closure (also know as Kleene-Star):* $L^* = \bigcup_{n \geq 0} L^n$
- *Positive closure:* $L^+ = \bigcup_{n > 0} L^n$

Let's look at some examples for these operations.

Example 5. *Let* $L_1 = \{1, 2, 3\}$ *and* $L_2 = \{3, 4, 5\}$
$L_1 \cup L_2 = \{1, 2, 3, 4, 5\}$
$L_2 \cap L_2 = \{3\}$
$L_1 - L_2 = \{1, 2\}$
$L_1^0 = \{\varepsilon\}$
$L_1^1 = L_1 = \{1, 2, 3\}$
$L_1^2 = \{11, 12, 13, 21, 22, 23, 31, 32, 33\}$
$L_1^* = \{\varepsilon, 1, 2, 3, 11, 12, 13, 21, 22, 23, 31, 32, 33, ...\}$
$L_1^+ = \{1, 2, 3, 11, 12, 13, 21, 22, 23, 31, 32, 33, ...\}$

Next, let's take a look at the classification of grammars. This classification is called Chomsky Hierarchy. These types of grammars are often referred as Type 0, 1, 2 and 3 grammar.

Definition 5. ***Chomsky Hierarchy*** *[Pet22, Hro11]*

1. *Every production* $\alpha \rightarrow \beta$ *with* $\alpha \in V^+$ *and* $\beta \in V^*$ *is a type 0 production.*
2. *A production* $\alpha \rightarrow \beta$ *is of type 1 iff* $\alpha, \beta \in V^+$ *and the length of* α *is not greater than the length of* β $(|\alpha| \leq |\beta|)$*. This grammar is called context-sensitive.*
3. *A production* $\alpha \rightarrow \beta$ *is of type 2 iff* $\alpha \in V_N$ *and* $\beta \in V^+$*. This grammar is called context-free.*
4. *A production* $\alpha \rightarrow \beta$ *is of type 3 iff* $\alpha \in V_N$ *and* $V_T^* \cdot V_N \cup V_T^*$ *This grammar is called regular.*

For i = 0, 1, 2, 3, a grammar is of type i if all its productions are of type i. For i = 0, 1, 2, 3, a language is of type i if it is generated by a type i grammar.

Since the definition for the different types i (i = 0, 1, 2, 3) is getting more restrictive by increasing i, each type $i + 1$ grammar is also a type i grammar. [Pet22] A type 0

grammar is a unrestricted grammar. [Cho59] According to Chomsky, a type 0 grammar is equally powerful as Turing Machines and type 3 grammars equally powerful as finite automatas.
The examples 1, 2, 3 which we have taken a look at, are context-free grammars. We can see, that grammar G_4 does not fulfill the requirements for a context-free grammar as defined in definition 5, since production 2, 3, 4, 5 and 6 violate the conditions. However, grammer G_4 fulfills the requirements of the definition for context-sensitive grammar (see definition 5).

Definition 6. *Epsilon Productions* *[Pet22]*
Given a grammar $G = (V_T, V_N, P, S)$ a prduction of the form $A \to \varepsilon$, where $A \in V_N$, is called an epsilon production.

Using an epsilon production, we are able to generate the empty word ε from the start symbol.

Definition 7. *Extended Grammar* *[Pet22]*
For i = 0, 1, 2, 3, an extended type i grammar is a grammar (V_T, V_N, P, S) whose set of productions P consists of productions of type i and, possibly, $n(\geq 1)$ epsilon productions of the form $A_1 \to \varepsilon$, ..., $A_n \to \varepsilon$, where the A_i's are distinct nonterminal symbols.

Using extended grammars, we are able to add empty words to our languages. We define the S-extended grammars as follows:

Definition 8. *S-extended Grammar* *[Pet22]*
For i = 0, 1, 2, 3, an S-extended type i grammar is a grammar (V_T, V_N, P, S) whose set of productions P consists of productions of type i and, possibly, the production $S \to \varepsilon$.

Using the S-extended grammar rules, we are also able to generate the empty word ε using the type i grammars, if the empty word is part of the corresponding language.

Theorem 1. *Relationship between S-extended Grammars and Extended Grammars* *[Pet22]*

1. *Every extended type 0 grammar is a type 0 grammar and vice versa.*
2. *Every extended type 1 grammar is a type 0 grammar.*
3. *Every extended type 2 grammar G such that $\varepsilon \notin L(G)$, there exists an equivalent type 2 grammar. For every extended type 2 grammar G such that $\varepsilon \in L(G)$, there exists an equivalent, S-extended type 2 grammar.*

4. *For every extended type 3 grammar G such that $\varepsilon \notin L(G)$, there exists an equivalent type 3 grammar. For every extended type 3 grammar G such that $\varepsilon \in L(G)$, there exists an equivalent, S-extended type 3 grammar.*

This definition defines the relationship between the S-extended grammars and the extended grammars.

Definition 9. ***Type 1, Context-Sensitive, Type 2 (or Context-Free), and Type 3 (or Regular) Production, Grammar and Language. Version with Epsilon Productions***[Pet22]

1. *Given a grammar $G = (V_T, V_N, P, S)$ we say that a production in P is of type 1 iff either it is of the form $\alpha \to \beta$, where $\alpha \in (V_T \cup N_N)^+$, and $|\alpha| \leq |\beta|$, or it is $S \to \varepsilon$, and if the production $S \to \varepsilon$ is in P, then the axiom S does not occur on the right hand side of any production in P.*

2. *Given a grammar $G = (V_T, V_N, P, S)$, we say that a production in P is context-sensitive iff either it is of the form $uAv \to uwv$, where $u, v \in V^*$, $A \in V_N$, and $w \in (V_T \cup V_N)^+$, or it is $S \in \varepsilon$, and if the production $S \to \varepsilon$ is in P, then the axiom S does not occur on the right hand side of any production in P.*

3. *Given a grammar $g = (V_T, V_N, P, S)$ we say that a production in P is of type 2 (or context-free) iff it is of the form $\alpha \to \beta$ where $\alpha \in V_N$ and $\beta \in V^*$.*

4. *Given a grammar $G = (V_T, V_N, P, S)$ we say that a production in P is of type 2 (or context-free) iff it is if the form $\alpha \to \beta$, where $\alpha \in V_N$ and $\beta \in V^*$.*

5. *Given a grammar $G = (V_T, V_N, P, S)$ we say that a production in P is of type 3 (or regular) iff it is of the form $\alpha \to \beta$, where $\alpha \in V_N$ and $\beta \in \{\varepsilon\} \cup V_T \cup V_T V_N$.*

A grammar is of type 1, context-sensitive, of type 2 (or context-free), and of type 3 (or regular) iff all its productions are of type 1, context-sensitive, of type 2 (or context-free) and of type 3 (or regular), respectively. A type 1, context-sensitive, type 2 (or context-free), and type 3 (or regular) language is a language generated by a type 1, context-sensitive, type 2 (or context-free), and type 3 (or regular) grammar respectively.

A.1.1 Finite Automata and Regular Grammars

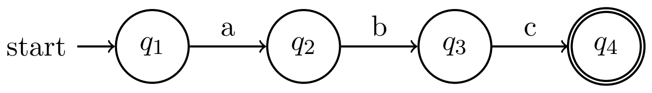
Regular expressions can be transformed to finite automata. These finite automata can be used, to generate words in the language as well as words, which are not in this language. Let's take a look at the definition of finite automata.

Definition 10. ***Deterministic Finite Automaton***[Pet22]
A deterministic finite automaton (also called finite automaton, for short) over the finite alphabet Σ (also called the input alphabet) is a quintuple $(Q, \Sigma, q_0, F, \delta)$ where

- *Q is a finite set of states,*
- *q_0 is an element of Q, called the initial state,*
- *$F \subseteq Q$ is the set of finial states, and*
- *δ is a total function, called the transition function, from $Q \times \Sigma$ to Q.*

Definition 11. ***Language Accepted by a finite Automaton***[Pet22]
We say that a finite automaton $(Q, \Sigma, q_0, F, \delta)$ accepts a word w in Σ^* iff $\delta^*(q_0, w) \in F$. A finite automaton accepts a language L iff it accepts every word in L and no other word. If a finite automaton M accepts the language L, we say that L is the language accepted by M. $L(M)$ denotes the language accepted by the finite automaton M.

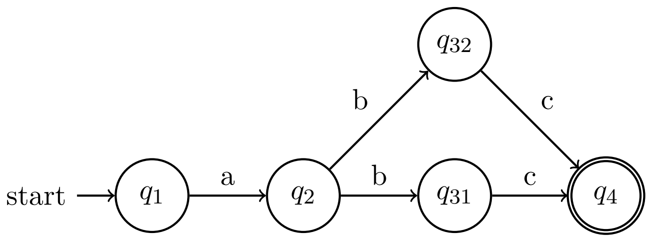
Consider this example of a deterministic finite automaton (also called state machine) for the language $L = \{abc\}$:



We can see in this example, that the state machine accepts all words in the language. We can also see, that the state machine does not accept any other words than the words in the language of the RegEx. By the given definition we can say that this state machine accepts the language $L = abc$.

Definition 12. ***Nondeterministic Finite Automaton***[Pet22]
A nondeterministic finite automaton is like a finite automaton, with the only difference that the transition function δ is a total function from $Q \times \Sigma$ to 2^Q, that is, from $Q \times \Sigma$ to the set of finite subsets of Q. Thus, the transition function δ returns a subset of states, rather than a single state.

Consider this example of a non-deterministic finite automata for the RegEx abc:



It shows a non-deterministic finite automata, accepting the same language as shown in the example before. The difference is, given the word *abc*, that there are multiple possible states receiving the input "*b*" in the state q_2.

Definition 13. ***Language Accepted by a Nondeterministic Finite Automaton****[Pet22]*
A nondeterministic finite automaton $(Q, \Sigma, q_0, F, \delta)$ *accepts a word* w *in* Σ^* *iff there exists a state in* $\delta^*(q_0, w)$ *which belongs to* F*. A nondeterministic finite automaton accepts a language* L *iff it accepts every word in* L *and no other word. If a nondeterministic finite automaton* M *accepts a language* L*, we say that* L *is the language accepted my* M*.*

Theorem 2. ***Equivalence Between S-extended Type 3 Grammars and Nondeterministic Finite Automata****[Pet22]*
For every S-extended type 3 grammar which generates the language $L \subseteq \Sigma^*$*, there exists a nondeterministic finite automaton over* Σ *which accepts* L *(and this automaton is said to be equivalent to that grammar) and vice versa.*

A.1.2 General definition of regular expressions

First of all, let's take a look at the basic definition of a regular expression.

Definition 14. ***Regular Expression****[Pet22]*
A regular expression over an alphabet Σ *is an expression* e *of the form:*
$e ::= \emptyset \mid a \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^*$
for any $a \in \Sigma$*.*

For simplicity, the concatenation of two regular expressions $e_1 \cdot e_2$ can be written as $e_1 e_2$. The empty regular expression is denoted using ε [Pet22]. The term regular expression is abbreviated either as RegEx or RExpr.

Definition 15. ***Language Denoted by a Regular Expression****[Pet22]*
A regular expression e *over the alphabet* Σ *denotes a language* $L(e) \subseteq \Sigma^*$ *which is defined by the following rules:*

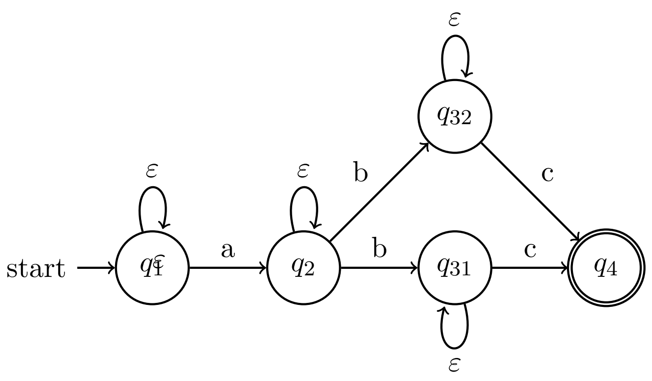
1. *$L(\emptyset) = \emptyset$*
2. *for any $a \in \Sigma, L(a) = a$*
3. *$L(e_1 \cdot e_2) = L(e_1) \cdot L(e_2)$, where on the left and side '$\cdot$' denotes concatenation of regular expressions, and on the right hand side '$\cdot$' denotes concatenation of languages as defined in definition 4.*
4. *$L(e_1 + e_2) = L(e_1) \cup L(e_2)$, and*
5. *$L(e^*) = (L(e))^*$, where on the right hand side '*' denotes the operation on the languages which is defined in definition 4.*

Definition 16. ***Equivalence Between Regular Expressions***[Pet22]
Two regular expression e_1 and e_2 are said to be equivalent, and we write $e_1 = e_2$, iff they denote the same language, that is, $L(e_1) = L(e_2)$.

Consider the following two regular expressions: $R_1 = a|b$ and $R_2 = b|a$. These two regular expressions are equal and we can write $R_1 = R_2$ because they denote the same language L, since $L(R_1) = a, b$ and $L(R_2) = b, a$.

Definition 17. ***RExpr Transition Graph***[Pet22]
An $RExpr_\Sigma$ transition graph $(Q, \Sigma, q_0, F, \delta)$ over the alphabet Σ is a multigraph like that of a nondeterministic finite automaton over Σ, except that the transition function δ is a total function from $Q \times RExpr_\Sigma$ to 2^Q such that for any $q \in Q$, $q \in \delta(q, \varepsilon)$.

Let's take a look at an example of an RExpr Transition Graph.



In this Transition Graph each node has a self-loop for the empty input epsilon.

Definition 18. ***Language Accepted by an RExpr Transition Graph***[Pet22]
An RExpr transition graph $(Q, \Sigma, q_0, F, \delta)$ accepts a word w in Σ^* iff there exists a state in $\delta^*(\{q_0\}, w)$, which belongs to F. An RExpr transition graph accepts a language L iff it accepts every word in L and no other word. If an RExpr transition graph T accepts a language L, we say that L is the language accepted by T.

Definition 19. ***Equivalence Between Regular Expressions, Finite Automata, Transition Graphs, and RExpr Transition Graphs***[Pet22]
(i) A regular expression and a finite automaton (or a transition graph, or an RExpr transition graph) are said to be equivalent iff the language denoted by the regular expression is the language accepted by the finite automaton (or the transition graph, or the RExpr transition graph, respectively).
Analougous definitions will be assumed for the notions of
(ii) the equivalence between finite automata and transition graphs (or RExpr transition graphs), and
(iii) the equivalence between transition graphs and RExpr transition graphs.

Theorem 3. ***Kleene Theorem***[Pet22]
(i) For every deterministic finite automaton D over the alphabet Σ there exists an equivalent regular expression over Σ, that is, a regular expression which denotes the language accepted by D.
(ii) For every regular expression e over the alphabet Σ there exists an equivalent deterministic finite automaton over Σ, that is, a finite automaton which accepts the language denoted by e.

The Kleene Theorem states, that there exists a finite automaton for every regular expression and vice versa. This gives us the opportunity, to use these automata to generate words in the language as well as words which are not in the language.

Theorem 4. ***Equivalence Between Regular Languages and RExpr***[Pet22]
A language is a regular language iff it is denoted by a regular expression.

Definition 20. ***Extended Regular Expressions***[Pet22]
An extended regular expression over the alphabet Σ is an expression e of the form:
$e ::= \emptyset \mid a \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^* \mid \overline{e} \mid e_1 \wedge e_2$
where a ranges over the alphabet Σ.

Definition 21. ***Language Denoted by an Extended Regular Expression****[Pet22]*
The language $L(e) \subseteq \Sigma^*$ denoted by an extended regular expression e over the alphabet Σ is defined by structural induction as follows:

- $L(\emptyset) = \emptyset$
- $L(a)\{a\}$ *for any* $a \in \Sigma$
- $L(e_1 \cdot e_2) = L(e_1) \cdot L(e_2)$
- $L(e_1 + e_2) = L(e_1) \cup L(e_2)$
- $L(e^*) = (L(e))^*$
- $L(\overline{e}) = \Sigma^* - L(e)$
- $L(e_1 \wedge e_2) = L(e_1) \cap L(e_2)$

Extended regular expressions are equivalent to regular expressions because regular expressions are closed under complementation and intersection.

A.2 Hoare Logic

To be able to reverse engineer static code and to find the precondition which an input has to match, we make use of the Hoare Logic and the corresponding Hoare Rules. In the following section, we revisit the first-order version of Hoare Logic and its rules. [Hoa69b]

Hoare Rules for Partial Correctness

At first, we revisit the notation of the Hoare Logic. The notation of the basic assertion is as follows

$$\{A\}\ p\ \{B\} \tag{A.1}$$

where A is the precondition, p is the program or statement and B is the postcondition. It describes that if the current state satisfies the precondition A, and the program p is executed and the code terminates, then the state will fulfill the postcondition B.

Empty Statement Rule

$$\{A\}\ skip\ \{A\} \tag{A.2}$$

The Empty Statement Rule states, that the skip statement or an empty statement does not change the precondition, meaning that the precondition holds after the statement as well.

Assignment Rule

$$\{A[x/b]\}\ x := b\ \{A\} \tag{A.3}$$

The Assignment Rule describes how an assignment is handled. If the precondition $A[x/b]$ (meaning that A with b substituted for x) holds before the assignment, and the statements assigns b to x, the postcondition B holds. Consider this example:

$$\{b = 4\}\ a := 4\ \{b = a\}$$

It means, that if the precondition $b = 4$ is fulfilled, and the program p ($a := 4$, meaning that the variable a is assigned the value 4) is executed, the postcondition $b = a$ is fulfilled.

Composition Rule

$$\frac{\{A\}\ p1\ \{B\} \quad \{B\}\ p2\ \{C\}}{\{A\}\ p1; p2\ \{C\}} \tag{A.4}$$

The Rule of Composition describes how two sequential statements are evaluated. If the precondition A holds before the execution, and p_1 and p_2 are executed sequentially, then the postcondition C holds. Consider the following example:

$$\frac{\{c+1=4\}\ a = c+1\ \{a=4\} \quad \{a=4\}\ b := a\ \{b=4\}}{\{c+1=4\}\ a = c+1; b := a\ \{b=4\}}$$

Conditional Rule

$$\frac{\{A \wedge b\}\ p\ \{C\} \quad \{A \wedge \neg b\}\ q\ \{C\}}{\{A\}\ \textbf{if}\ \text{b}\ \textbf{then}\ \text{p}\ \textbf{else}\ \text{q}\ \textbf{end}\ \{C\}} \tag{A.5}$$

The Rule of Condition describes how an If-Statement is evaluated. When executing the If-statement, depending on the condition b, either the statement p or q is executed. Consider this example:

$$\frac{\{x \geq 0\}\ \mathit{skip}\ \{x \geq 0\} \quad \{x < 0\}\ x := -x\ \{x \geq 0\}}{\{x \geq 0 \vee x < 0\}\ \textbf{if}\ x \geq 0\ \textbf{then}\ \mathit{skip}\ \textbf{else}\ x := -x\ \textbf{end}\ \{x \geq 0\}}$$

Iteration Rule

$$\frac{\{A \wedge b\}\ p\ \{C\}}{\{A\}\ \textbf{while}\ b\ \textbf{do}\ p\ \textbf{end}\ \{\neg b \wedge C\}} \tag{A.6}$$

The Rule of Iteration describes how a While-Loop-Statement is evaluated. When executing the While-statement, depending on the condition b, the statement p is executed until the condition does not evaluate to true anymore. The iteration rule won't be used in this work and is listed for the sake of completeness.

Consequence Rule

$$\frac{A' \implies A, \{A\}\ p\ \{C\}, C \implies C'}{\{A'\}\ p\ \{C'\}} \tag{A.7}$$

The Rule of Consequence describes how we can deal with conditions, which are stronger or softer compared to the conditions needed. If the precondition A' implies A, and the precondition C implies C', and the triple $\{A\}\ p\ \{C\}$ holds, then $\{A'\}\ p\ \{C'\}$ holds. Consider this example:

$$\frac{true \implies 1 > 0, \{1 > 0\}\ x := 1\ \{x > 0\}}{\{true\}\ x := 1\ \{x > 0\}}$$

A.3 Logic Method Mappers

$$[\![t.\texttt{codePointCount}(int\ sIdx, int\ eIdx) == c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & c < 0 \\ & \quad \vee c < eIdx - sIdx \\ & \quad \vee c > 2(eIdx - sIdx) \\ .\{\mathrm{eIdx} - \mathrm{sIdx}\}recursion.* \text{ where recursion} = \{x_1 x_2 \dots x_n | & \text{otherwise} \\ \quad x = [\texttt{\textbackslash uD800} - \texttt{\textbackslash uDBFF}][\texttt{\textbackslash uDC00} - \texttt{\textbackslash uDFFF}] \text{ with value=2 or} & \\ \quad x = [\texttt{\textbackslash u0000} - \texttt{\textbackslash uD7FF}\texttt{\textbackslash uE000} - \texttt{\textbackslash uFFFF}] \text{ with value=1 and} & \\ \quad \sum(value) = \mathrm{c} \text{ and } n = \mathrm{eIdx} - \mathrm{sIdx}\} & \end{cases}$$

$$[\![t.\texttt{codePointCount}(int\ sIdx, int\ eIdx) \neq c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & c = 0 \vee eIdx - sIdx = 0 \\ .\{\mathrm{sIdx},\} & c \neq 0 \wedge eIdx - sIdx = 0 \\ .\{\mathrm{eIdx}\}.* & c < eIdx - sIdx \\ .\{\mathrm{eIdx} - \mathrm{sIdx}\}recursion.* \text{ where recursion} = \{x_1 x_2 \dots x_n | & \text{otherwise} \\ \quad x = [\texttt{\textbackslash uD800} - \texttt{\textbackslash uDBFF}][\texttt{\textbackslash uDC00} - \texttt{\textbackslash uDFFF}] \text{ with value=2 or} & \\ \quad x = [\texttt{\textbackslash u0000} - \texttt{\textbackslash uD7FF}\texttt{\textbackslash uE000} - \texttt{\textbackslash uFFFF}] \text{ with value=1 and} & \\ \quad \sum(value) \neq \mathrm{c} \text{ and } n = \mathrm{eIdx} - \mathrm{sIdx}\} & \end{cases}$$

$$[\![t.\texttt{codePointCount}(int\ sIdx, int\ eIdx) < c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & c \leq eIdx - sIdx \\ .\{\mathrm{sIdx},\} & eIdx - sIdx = 0 \\ & \quad \wedge c > 0 \\ .\{\mathrm{sIdx}\}recursion.* \text{ where recursion} = \{x_1 x_2 \dots x_n | & \text{otherwise} \\ \quad x = [\texttt{\textbackslash uD800} - \texttt{\textbackslash uDBFF}][\texttt{\textbackslash uDC00} - \texttt{\textbackslash uDFFF}] \text{ with value=2 or} & \\ \quad x = [\texttt{\textbackslash u0000} - \texttt{\textbackslash uD7FF}\texttt{\textbackslash uE000} - \texttt{\textbackslash uFFFF}] \text{ with value=1 and} & \\ \quad \sum(value) < \mathrm{c} \text{ and } n = \mathrm{eIdx} - \mathrm{sIdx}\} & \end{cases}$$

$$[\![t.\texttt{codePointCount}(\mathit{int}\ sIdx, \mathit{int}\ eIdx) > c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} \text{?!} & c \geq 2(eIdx - sIdx) \\ .\{\mathrm{eIdx}\}.* & c < (eIdx - sIdx) \\ .\{\mathrm{sIdx}\}.\{\mathrm{eIdx} - \mathrm{sIdx}\}.* & c = 0 \\ .\{\mathrm{sIdx}\}\mathit{recursion}.* \text{ where recursion } = \{x_1 x_2 \ldots x_n | & \text{otherwise} \\ \quad x = [\texttt{\textbackslash uD800} - \texttt{\textbackslash uDBFF}][\texttt{\textbackslash uDC00} - \texttt{\textbackslash uDFFF}] \text{ with value=2 or} & \\ \quad x = [\texttt{\textbackslash u0000} - \texttt{\textbackslash uD7FF}\texttt{\textbackslash uE000} - \texttt{\textbackslash uFFFF}] \text{ with value=1 and} & \\ \quad \sum(\mathit{value}) > \mathrm{c} \text{ and } n = \mathrm{eIdx} - \mathrm{sIdx}\} & \end{cases}$$

$$[\![t.\texttt{compareTo}(\mathit{String}\ aStr) == c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} \text{?!} & c < 0 \wedge len(aStr) = 0 \\ \mathrm{aStr} & c = 0 \\ .\{\mathrm{c}\} & c > 0 \wedge len(aStr) = 0 \\ .\{\mathrm{c}\} | \Big|_{n=0}^{\mathrm{c}-1} \Big(\mathrm{aStr}[0:n]\texttt{\textbackslash x}\{\mathrm{aStr}[\mathrm{n}] + \mathrm{c}\}.* \Big) & c > 0 \wedge \neg len(aStr) = 0 \\ \mathrm{aStr}[0 : len(\mathrm{aStr}) + \mathrm{c}] | \Big|_{n=0}^{\mathrm{c}-1} \Big(\mathrm{aStr}[0:n]\texttt{\textbackslash x}\{\mathrm{aStr}[\mathrm{n}] + \mathrm{c}\}.* \Big) & c < 0 \\ & \wedge len(aStr) + c \geq 0 \\ \Big|_{n=0}^{\mathrm{c}-1} \Big(\mathrm{aStr}[0:n]\texttt{\textbackslash x}\{\mathrm{aStr}[\mathrm{n}] + \mathrm{c}\}.* \Big) & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{compareTo}(\mathit{String}\ aStr) \neq c]\!]_{\mathrm{RE}} \rightarrow t.\texttt{compareTo}(aStr) < c | t.\texttt{compareTo}(aStr) > c$$

$$[\![t.\texttt{compareTo}(\mathit{String}\ aStr) < c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} \mathit{shorterRestrictions}(\mathrm{aStr}, \mathrm{c}) & \neg len(aStr) = 0 \wedge c < 1 \\ \quad | \Big|_{n=0}^{len(\mathrm{aStr})} \Big(\mathrm{aStr}[0:n][\texttt{\textbackslash x}\{0\} - \texttt{\textbackslash x}\{(\mathrm{aStr}[\mathrm{n}] + \mathrm{c} - 1)\}].* \Big) & \\ \mathit{shorterRestrictions}(\mathrm{aStr}, \mathrm{c}) & \neg len(aStr) = 0 \wedge c = 1 \\ \quad | \Big|_{n=0}^{len(\mathrm{aStr})} \Big(\mathrm{aStr}[0:n][\texttt{\textbackslash x}\{0\} - \texttt{\textbackslash x}\{(\mathrm{aStr}[\mathrm{n}] - 1)\}].* \Big) & \\ \mathit{shorterRestrictions}(\mathrm{aStr}, \mathrm{c}) & \neg len(aStr) = 0 \wedge c > 1 \\ \quad | \Big|_{n=0}^{len(\mathrm{aStr})} \Big(\mathrm{aStr}[0:n][\texttt{\textbackslash x}\{0\} - \texttt{\textbackslash x}\{\mathrm{aStr}[\mathrm{n}] - 1\}].* \Big) & \\ \quad | \Big|_{n=0}^{len(\mathrm{aStr})} \Big(\mathrm{aStr}[0:n][\texttt{\textbackslash x}\{\mathrm{aStr}[\mathrm{n}] + 1\} - \texttt{\textbackslash x}\{\mathrm{aStr}[\mathrm{n}] + \mathrm{c} - 1\}].* \Big) & \end{cases}$$

$$\mathit{shorterRestrictions}(aStr, c) \rightarrow$$

$$\begin{cases} \Big|_{n=0}^{len(\mathrm{aStr})-1} \Big(\mathrm{aStr}[0:n] \Big) | \mathrm{aStr} | \mathrm{aStr}.\{0,\mathrm{c} - 1\} & c > 0 \\ \Big|_{n=0}^{len(\mathrm{aStr})-1} \Big(\mathrm{aStr}[0:n] \Big) & c = 0 \\ \Big|_{n=0}^{len(\mathrm{aStr})+\mathrm{c}-1} \Big(\mathrm{aStr}[0:n] \Big) & c < 0 \wedge len(aStr) > -c \\ () & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{compareTo}(String\ aStr) > c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases}
\mathrm{aStr}.\{\ \max(\mathrm{c}+1,0),\} & \\
\quad | \mathop{\bigg|}\limits_{n=\max(-len(\mathrm{aStr}),\mathrm{c}+1)}^{0} \left(\mathrm{aStr}[0:len(\mathrm{aStr})+n]\right) & c<-1 \wedge len(aStr)=0 \\
\mathrm{aStr}.\{\ \max(\mathrm{c}+1,0),\} & c<-1 \wedge \neg len(aStr)=0 \\
\quad | \mathop{\bigg|}\limits_{n=\max(-len(\mathrm{aStr}),\mathrm{c}+1)}^{0} \left(\mathrm{aStr}[0:len(\mathrm{aStr})+n]\right) & \\
\quad | generateCompareGreater(\mathrm{aStr},\mathrm{c}) & \\
generateCompareGreater(\mathrm{aStr},\mathrm{c}) & \neg len(aStr)=0 \\
\mathrm{aStr}.\{\ \max(\mathrm{c}+1,0),\} & \text{otherwise}
\end{cases}$$

$$generateCompareGreater(aStr,c) \rightarrow$$

$$\begin{cases}
\mathop{\bigg|}\limits_{n=0}^{len(\mathrm{aStr})} \left(\mathrm{aStr}[0:n][\backslash\texttt{x}\{\mathrm{aStr}[\mathrm{n}]+\mathrm{c}+1\}-\backslash\texttt{x}\{\texttt{FFFF}\}].*\right) & c \geq 0 \\
\mathop{\bigg|}\limits_{n=0}^{len(\mathrm{aStr})} (\mathrm{aStr}[0:n][\backslash\texttt{x}\{\mathrm{aStr}[\mathrm{n}]+1\}-\backslash\texttt{x}\{\texttt{FFFF}\}].*) & c=-1 \\
\mathop{\bigg|}\limits_{n=0}^{len(\mathrm{aStr})} \left(\mathrm{aStr}[0:n]\backslash\texttt{x}\{\mathrm{aStr}[\mathrm{n}]+\mathrm{c}+1\}-\backslash\texttt{x}\{\mathrm{aStr}[\mathrm{n}]-1\}].*\right) & \text{otherwise} \\
\quad | \mathop{\bigg|}\limits_{n=0}^{len(\mathrm{aStr})} \left(\mathrm{aStr}[0:n][\backslash\texttt{x}\{\mathrm{aStr}[\mathrm{n}]+1\}-\backslash\texttt{x}\{\texttt{FFFF}\}].*\right) &
\end{cases}$$

$$[\![t.\texttt{contains}(String\ aStr) == c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases}
.*\mathrm{aStr}.* & c = true \\
\hat{}\,?!(.*\mathrm{aStr}.*\$).*\$ & \text{otherwise}
\end{cases}$$

$$[\![t.\texttt{indexOf}(char\ char, int\ fIdx) == c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases}
?! & (char<0 \vee char>MaxCodePoint) \\
 & \quad \wedge c \neq -1 \\
.* & (char<0 \vee char>MaxCodePoint) \\
 & \quad \wedge c = -1 \\
?! & (c>-1 \wedge fIdx>c) \vee c<-1 \\
[\hat{}\backslash\texttt{x}\{\mathrm{char}\}]* & c=-1 \\
(.\{\mathrm{fIdx}\}[\hat{}\backslash\texttt{x}\{\mathrm{char}\}]\{\mathrm{c}-\mathrm{fIdx}\}\backslash\texttt{x}\{\mathrm{char}\}.*) & \text{otherwise}
\end{cases}$$

$$[\![t.\texttt{indexOf}(char\ char, int\ fIdx) \neq c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases}
.* & (char<0 \vee char>MaxCodePoint) \\
 & \quad \wedge c \neq -1 \\
?! & (char<0 \vee char>MaxCodePoint) \\
 & \quad \wedge c = -1 \\
.\{\mathrm{fIdx}\}.*[\backslash\texttt{x}\{\mathrm{char}\}.*] & c=-1 \\
.* & c<-1 \vee c<fIdx \\
(.\{\ \max(0,\mathrm{c})\}[\hat{}\backslash\texttt{x}\{\mathrm{char}\}].*)|.\{\texttt{0},\max(0,\mathrm{c})\} & \text{otherwise}
\end{cases}$$

$$[\![t.\texttt{indexOf}(\mathit{char}\ \mathit{char}, \mathit{int}\ fIdx) < c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & c <= -1 \\ .* & (\mathit{char} < 0 \vee \mathit{char} > MaxCodePoint) \wedge c \geq 0 \\ .\{\texttt{0},\mathrm{fIdx}\}|.\{\mathrm{fIdx}\}[\hat{}\backslash\texttt{x}\{\mathrm{char}\}]* & fIdx \geq c \vee c = 0 \\ .\{\texttt{0},\mathrm{fIdx}\}|.\{\mathrm{fIdx}\}[\hat{}\backslash\texttt{x}\{\mathrm{char}\}]* & \text{otherwise} \\ \quad |.\{\mathrm{fIdx}\}.\{\texttt{0},\mathrm{c} - \mathrm{fIdx} - 1\}\backslash\texttt{x}\{\mathrm{char}\}.* & \end{cases}$$

$$[\![t.\texttt{indexOf}(\mathit{char}\ \mathit{char}, \mathit{int}\ fIdx) > c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & c < -1 \\ ?! & c \geq -1 \wedge (\mathit{char} < 0 \\ & \quad \vee \mathit{char} > MaxCodePoint) \\ .\{\mathrm{fIdx}\}. * [\backslash\texttt{x}\{\mathrm{char}\}.*] & c = -1 \\ (.\{\mathrm{fIdx}\}[\hat{}\backslash\texttt{x}\{\mathrm{char}\}]\{\max(0, \mathrm{c} - \mathrm{fIdx} + 1), \}[\backslash\texttt{x}\{\mathrm{char}\}].*) & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{indexOf}(\mathit{String}\ \mathit{str}, \mathit{int}\ fIdx) == c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .\{\mathrm{c}\} & (len(str) \wedge c < fIdx \wedge c \geq 0) \\ ?! & c < -1 \vee (c > -1 \wedge fIdx > c) \\ & \quad \vee (c = -1 \wedge len(str) = 0) \\ (.\{\texttt{0},\mathrm{fIdx}\})|(.\{\mathrm{fIdx}\}((?!\mathrm{str}).)*) & c = -1 \\ .\{\mathrm{fIdx}\}((?!\mathrm{str}).)\{\mathrm{c} - \mathrm{fIdx}\}\mathrm{str}.* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{indexOf}(\mathit{String}\ \mathit{str}, \mathit{int}\ fIdx) \neq c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .\{\max(0, \mathrm{c} + 1), \} & len(str) = 0 \wedge c < fIdx \\ & \quad \wedge c \leq 0 \\ .\{\max(0, \mathrm{c} + 1), \}|.\{\texttt{0},\mathrm{c} + 1\} & len(str) = 0 \wedge c < fIdx \\ & \quad \wedge c > 0 \\ .* & len(str) = 0 \wedge c < -1 \\ .\{\texttt{1}, \} & len(str) = 0 \wedge c = -1 \\ ?! & len(str) = 0 \wedge fIdx = 0 \\ & \quad \wedge c = 0 \\ .\{\texttt{0}, \max(0, \mathrm{c} - 1)\} & len(str) = 0 \wedge c = fIdx \\ .\{\mathrm{fIdx}\}.*\mathrm{str}.* & c = -1 \\ .* & c < -1 \vee c < fIdx \\ (.\{\mathrm{c}\}((?!\mathrm{str}).)). * |.\{\texttt{0},\mathrm{c}\} & \text{otherwise} \\ \quad \Big| \Big|_{n=\mathrm{fIdx}}^{\mathrm{c}-len(\mathrm{str})} \left(.\{\mathrm{fIdx}\}\mathrm{str}.\{\mathrm{c} - len(\mathrm{str}) - n + 1\}.* \right) & \end{cases}$$

$$[\![t.\texttt{indexOf}(String\ str,int\ fIdx) <c]\!]_{\mathrm{RE}} \rightarrow$$
$$\begin{cases}
?! & c \leq -1 \vee (len(str) = 0 \wedge c = 0) \\
.* & len(str) = 0 \wedge c = 1 \wedge fIdx \leq 0 \\
.\{0,0\} & len(str) = 0 \wedge c = 1 \wedge fIdx > 0 \\
?! & len(str) = 0 \wedge c < 0 \\
.* & len(str) = 0 \wedge c > 1 \wedge fIdx < c \\
.\{0,\mathrm{c}-1\} & len(str) = 0 \wedge c > 1 \wedge fIdx \geq c \\
.\{0,\mathrm{fIdx}\}|(.\{\mathrm{fIdx}\}((?!\mathrm{str}).)*) & fIdx \geq c \vee c = 0 \\
(.\{\mathrm{fIdx}\}.\{0,\mathrm{c}-\mathrm{fIdx}-1\}\mathrm{str}.*)|(.\{0,\mathrm{fIdx}\}) & c \neq 0 \\
\quad |(.\{\mathrm{fIdx}\}((?!\mathrm{str}).)*) &
\end{cases}$$

$$[\![t.\texttt{indexOf}(String\ str,int\ fIdx) >c]\!]_{\mathrm{RE}} \rightarrow$$
$$\begin{cases}
?! & len(str) = 0 \wedge c \geq fIdx \\
.\{\max(0,\mathrm{c}+1),\} & len(str) = 0 \wedge c < fIdx \\
.* & c < -1 \\
.\{\mathrm{fIdx}\}.*\mathrm{str}.* & c = -1 \\
.\{\mathrm{fIdx}\}(((?!\mathrm{str}).)\{\max(0,\mathrm{c}-\mathrm{fIdx}+1),\})\mathrm{str}.* & c > -1
\end{cases}$$

$$[\![t.\texttt{isBlank}() ==c]\!]_{\mathrm{RE}} \rightarrow$$
$$\begin{cases}
((\backslash\mathtt{x}\{9\})|(\backslash\mathtt{x}\{a\})|(\backslash\mathtt{x}\{b\})|(\backslash\mathtt{x}\{c\})|(\backslash\mathtt{x}\{d\})|(\backslash\mathtt{x}\{1c\}) & c = true \\
\quad |(\backslash\mathtt{x}\{1d\})|(\backslash\mathtt{x}\{1e\})|(\backslash\mathtt{x}\{1f\})|(\backslash\mathtt{x}\{20\})|(\backslash\mathtt{x}\{1680\}) & \\
\quad |(\backslash\mathtt{x}\{2000\})|(\backslash\mathtt{x}\{2001\})|(\backslash\mathtt{x}\{2002\})|(\backslash\mathtt{x}\{2003\}) & \\
\quad |(\backslash\mathtt{x}\{2004\})|(\backslash\mathtt{x}\{2005\})|(\backslash\mathtt{x}\{2006\})|(\backslash\mathtt{x}\{2008\}) & \\
\quad |(\backslash\mathtt{x}\{2009\})|(\backslash\mathtt{x}\{200a\})|(\backslash\mathtt{x}\{2028\})|(\backslash\mathtt{x}\{2029\})|(\backslash\mathtt{x}\{205f\})|(\backslash\mathtt{x}\{3000\}))* & \\
\hat{}?!(((\backslash\mathtt{x}\{9\})|(\backslash\mathtt{x}\{a\})|(\backslash\mathtt{x}\{b\})|(\backslash\mathtt{x}\{c\})|(\backslash\mathtt{x}\{d\})|(\backslash\mathtt{x}\{1c\}) & \text{otherwise} \\
\quad |(\backslash\mathtt{x}\{1d\})|(\backslash\mathtt{x}\{1e\})|(\backslash\mathtt{x}\{1f\})|(\backslash\mathtt{x}\{20\})|(\backslash\mathtt{x}\{1680\}) & \\
\quad |(\backslash\mathtt{x}\{2000\})|(\backslash\mathtt{x}\{2001\})|(\backslash\mathtt{x}\{2002\})|(\backslash\mathtt{x}\{2003\}) & \\
\quad |(\backslash\mathtt{x}\{2004\})|(\backslash\mathtt{x}\{2005\})|(\backslash\mathtt{x}\{2006\})|(\backslash\mathtt{x}\{2008\})|(\backslash\mathtt{x}\{2009\}) & \\
\quad |(\backslash\mathtt{x}\{200a\})|(\backslash\mathtt{x}\{2028\})|(\backslash\mathtt{x}\{2029\})|(\backslash\mathtt{x}\{205f\})|(\backslash\mathtt{x}\{3000\}))*\$).*\$ &
\end{cases}$$

$$[\![t.\texttt{lastIndexOf}(char\ char) ==c]\!]_{\mathrm{RE}} \rightarrow$$
$$\begin{cases}
?! & (c > -1 \wedge (char < 0 \vee char > MaxCodePoint)) \\
 & \quad \vee (c < -1) \\
.* & c = -1 \wedge (char < 0 \vee char > MaxCodePoint) \\
[\hat{}\backslash\mathtt{x}\{\mathrm{char}\}]* & c = -1 \wedge \neg(char < 0 \vee char > MaxCodePoint) \\
.\{\mathrm{c}\}[\backslash\mathtt{x}\{\mathrm{char}\}][\hat{}\backslash\mathtt{x}\{\mathrm{char}\}]* & \text{otherwise}
\end{cases}$$

$$[\![t.\texttt{lastIndexOf}(char\ char) \neq c]\!]_{\mathrm{RE}} \rightarrow$$
$$\begin{cases}
?! & c = -1 \wedge (char < 0 \vee char > MaxCodePoint) \\
.*[\backslash\mathtt{x}\{\mathrm{char}\}].* & c = -1 \wedge \neg(char < 0 \vee char > MaxCodePoint) \\
.* & c < -1 \vee (char < 0 \vee char > MaxCodePoint) \\
(.\{0,\mathrm{c}\})|(.\{\mathrm{c}\}[\hat{}\backslash\mathtt{x}\{\mathrm{char}\}].*) & \text{otherwise} \\
\quad |(.\{\mathrm{c}\}[\backslash\mathtt{x}\{\mathrm{char}\}].*[\backslash\mathtt{x}\{\mathrm{char}\}].*) &
\end{cases}$$

$$[\![t.\texttt{lastIndexOf}(\mathit{char}\ \mathit{char}) <c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & c \leq -1 \\ .* & (\mathit{char} < 0 \vee \mathit{char} > MaxCodePoint) \\ & \quad \wedge c > -1 \\ (.\{\texttt{0}, \max(0, \mathrm{c}-1)\}[\backslash\texttt{x}\{\mathrm{char}\}]) & c > 0 \\ \quad |(.\{\texttt{0}, \max(0, \mathrm{c}-1)\}[\backslash\texttt{x}\{\mathrm{char}\}][\hat{}\backslash\texttt{x}\{\mathrm{char}\}]*) & \\ \quad |([\hat{}\backslash\texttt{x}\{\mathrm{char}\}]*) & \\ [\hat{}\backslash\texttt{x}\{\mathrm{char}\}]* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(\mathit{char}\ \mathit{char}) >c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & c < -1 \\ ?! & c \geq -1 \wedge (\mathit{char} < 0 \vee \mathit{char} > MaxCodePoint) \\ .\{\max(0, \mathrm{c}+1),\}[\backslash\texttt{x}\{\mathrm{char}\}].* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(\mathit{char}\ \mathit{char},\mathit{int}\ fIdx) ==c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & fIdx < 0 \wedge c \neq -1 \\ .* & ((\mathit{char} < 0 \vee \mathit{char} > MaxCodePoint) \\ & \quad \wedge c = -1) \vee (fIdx < 0 \wedge c = -1) \\ ?! & (\mathit{char} < 0 \vee \mathit{char} > MaxCodePoint) \\ & \quad \wedge c \neq -1 \\ ?! & c > fIdx \vee c < -1 \\ ([\hat{}\backslash\texttt{x}\{\mathrm{char}\}]\{\texttt{0},\mathrm{fIdx}\})|([\hat{}\backslash\texttt{x}\{\mathrm{char}\}]\{\mathrm{fIdx}+1\}.*) & c = -1 \\ (.\{\mathrm{c}\}[\backslash\texttt{x}\{\mathrm{char}\}][\hat{}\backslash\texttt{x}\{\mathrm{char}\}]\{\texttt{0}, \max(0, \mathrm{fIdx}-1)\}) & \text{otherwise} \\ \quad |(.\{\mathrm{c}\}[\backslash\texttt{x}\{\mathrm{char}\}][\hat{}\backslash\texttt{x}\{\mathrm{char}\}]\{\mathrm{fIdx}-\mathrm{c}\}.*) & \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(\mathit{char}\ \mathit{char},\mathit{int}\ fIdx) \neq c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & c < -1 \\ ?! & (\mathit{char} < 0 \vee \mathit{char} > MaxCodePoint) \\ & \quad \vee (fIdx < 0) \wedge c = -1 \\ \Bigg|_{n=0}^{\mathrm{fIdx}} \left(.\{n\}[\backslash\texttt{x}\{\mathrm{char}\}].*\right) & c = -1 \\ .* & ((\mathit{char} < 0 \vee \mathit{char} > MaxCodePoint) \\ & \quad \vee c > fIdx) \wedge c \geq 0 \\ .* & c \geq 0 \wedge fIdx < 0 \\ (.\{\texttt{0},\mathrm{c}\})|(.\{\mathrm{c}\}[\hat{}\backslash\texttt{x}\{\mathrm{char}\}].*) & \text{otherwise} \\ \quad | \Bigg|_{n=0}^{\mathrm{fIdx}-\mathrm{c}-1} \left(.\{\mathrm{c}\}[\backslash\texttt{x}\{\mathrm{char}\}].\{\mathrm{n}\}[\backslash\texttt{x}\{\mathrm{char}\}].*\right) & \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(char\ char, int\ fIdx)\ {<}c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} \text{?!} & c \leq -1 \\ .* & (char < 0 \vee char > MaxCodePoint) \\ & \quad \vee (c \geq 0 \wedge fIdx < 0) \\ (.\{0,c-1\}[\backslash x\{char\}])|([\hat{}\backslash x\{char\}]\{0,fIdx+1\}) & \text{otherwise} \\ \quad |([\hat{}\backslash x\{char\}]\{fIdx+1\}.*) & \\ \quad | \Big|_{n=0}^{c-1} \Big(.\{n\}[\backslash x\{char\}][\hat{}\backslash x\{char\}]\{0, \max(0, fIdx - n)\}\Big) & \\ \quad | \Big|_{n=0}^{c-1} \Big(.\{n\}[\backslash x\{char\}][\hat{}\backslash x\{char\}]\{\max(0, fIdx - n)\}.*\Big) & \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(char\ char, int\ fIdx)\ {>}c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & c < -1 \\ \text{?!} & fIdx \leq c \vee (char < 0 \vee char > MaxCodePoint) \\ .\{c+1,fIdx\}[\backslash x\{char\}].* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str)\ {==}c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} \text{?!} & (c < -1) \vee (c = -1 \wedge len(str) = 0) \\ (\text{?!}(str).)* & c = -1 \wedge len(str) > 0 \\ .\{c\} & c > -1 \wedge len(str) = 0 \\ .\{c\}str((\text{?!}str).)* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str)\ {\neq}c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & (c = -1 \wedge len(str) = 0) \vee (c < -1) \\ .*str.* & c = -1 \wedge len(str) \neq 0 \\ .\{c+1,\} & c > -1 \wedge len(str) = 0 \\ (.\{0,c-1\})|(.\{c+1,\}) & c > -1 \wedge c \neq 0 \wedge len(str) = 0 \\ (.\{0,c\})|(.\{c\}(\text{?!}(str)).*)|(.\{c\})str.*str.* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str)\ {<}c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} \text{?!} & (c = -1) \vee (c = 0 \wedge len(str) = 0) \\ (.\{0, \max(0, c-1)\}str)|(.\{0, \max(0, c-1)\}str(\text{?!}(str).)*) & c > 0 \\ \quad |((\text{?!}(str).)*) & \\ ((\text{?!}(str).)*) & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str)\ {>}c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & c < -1 \\ .\{\max(0, c+1),\}str.* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str, int\ fIdx) == c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & fIdx < 0 \land c \neq -1 \\ .* & fIdx < 0 \land c = -1 \\ ?! & (c > fIdx \land c < -1) \\ & \quad \lor (c = -1 \land len(str) = 0) \\ (((?!\mathrm{str}).)\{0,\mathrm{fIdx}\})|((((?!\mathrm{str}).)\{\mathrm{fIdx}+1\}).*) & c = -1 \land len(str) \neq 0 \\ (.\{\mathrm{c}\}\mathrm{str}(?!(\mathrm{str}).)\{0, \max(0, \mathrm{fIdx} - \mathrm{len}(\mathrm{str}))\}) & \text{otherwise} \\ \quad |(.\{\mathrm{c}\}\mathrm{str}(?!(\mathrm{str}).)\{\mathrm{fIdx} - \mathrm{c}\}.*) & \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str, int\ fIdx) \neq c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & c < -1 \\ ?! & c = -1 \land fIdx < 0 \\ .\{0,\mathrm{fIdx}\}\mathrm{str}.* & c = -1 \land fIdx \geq 0 \\ .* & c \geq 0 \land (c > fIdx \lor fIdx < 0) \\ ?! & c \geq 0 \land c \leq fIdx \land len(str) = 0 \\ & \quad \land fIdx = 0 \land c = 0 \\ .\{\mathrm{c}+1,\} & c \geq 0 \land c \leq fIdx \land len(str) = 0 \land c = 0 \\ .\{0,\mathrm{fIdx}-1\} & c \geq 0 \land c \leq fIdx \land len(str) = 0 \land fIdx = c \\ (.\{0,\mathrm{c}-1\})|(.\{\mathrm{c}+1,\}) & c \geq 0 \land c \leq fIdx \land len(str) = 0 \\ (.\{0,\mathrm{c}\})|(.\{\mathrm{c}\}(?!(\mathrm{str}).).*) & \text{otherwise} \\ \quad |(.\{\mathrm{c}\}\mathrm{str}.\{0,\mathrm{fIdx} - \mathrm{c} - len(\mathrm{str})\}\mathrm{str}.*) & \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str, int\ fIdx) < c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} ?! & c \leq 1 \\ .* & c \geq 0 \land fIdx < 0 \\ ?! & len(str) = 0 \land c = 0 \\ .* & len(str) = 0 \land c > fIdx \\ .\{0,\mathrm{c}-1\}\mathrm{str} & len(str) = 0 \\ (.\{0,\mathrm{c}-1\}\mathrm{str})|((?!(\mathrm{str}).)\{0,\mathrm{fIdx}+1\}) & \text{otherwise} \\ \quad |((?!(\mathrm{str}).)\{\mathrm{fIdx}+1\}.*) & \\ \quad | \Big|_{n=0}^{\mathrm{c}-1} \left(.\{\mathrm{n}\}\mathrm{str}(?!(\mathrm{str}).)\{0, \max(0, \mathrm{fIdx} - n - len(\mathrm{str}))\} \right) & \\ \quad | \Big|_{n=0}^{\mathrm{c}-1} \left(.\{\mathrm{n}\}\mathrm{str}(?!(\mathrm{str}).)\{ \max(0, \mathrm{fIdx} - n - len(\mathrm{str}))\}.* \right) & \end{cases}$$

$$[\![t.\texttt{lastIndexOf}(String\ str, int\ fIdx) > c]\!]_{\mathrm{RE}} \rightarrow$$

$$\begin{cases} .* & c < -1 \\ ?! & fIdx \leq c \\ .\{\mathrm{c}+1,\mathrm{fIdx}\}\mathrm{str}.* & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{length}()\ \neq c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} .\{\texttt{1},\} & c = 0 \\ .* & c < 0 \\ .\{\texttt{0},\text{c}-1\}|.\{\text{c}+1,\} & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{length}()\ < c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} ?! & c <= 0 \\ .\{\texttt{0},\text{c}-1\} & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{length}()\ > c]\!]_{\text{RE}} \rightarrow .\{\max(0,\text{c}+1),\}$$

$$[\![t.\texttt{matches}(regex)\ == c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} \texttt{regex} & c = true \\ \hat{}?!(\text{regex}\$).*\$ & \text{otherwise} \end{cases}$$

$$[\![t.\texttt{offsetByCodePoints}(int\ idx, int\ offset)\ == c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} ?! & offset > 2(c - idx) \\ & \quad \vee offset < c \\ .\{\text{idx}\}recursion.* \text{ where recursion } = \{x_1 x_2 \dots x_n | & \text{otherwise} \\ \quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \\ \quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\ \quad \sum(value) = \text{offset and } n = \text{c} - \text{idx}\} & \end{cases}$$

$$[\![t.\texttt{offsetByCodePoints}(int\ idx, int\ offset) \neq c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases}
?! & offset = 0 \wedge c = idx \\
.\{\text{idx},\} & offset = 0 \wedge c \neq idx \\
.\{\text{idx}\}recursion.* \text{ where recursion } = \{x_1x_2\dots x_n| & offset > 0 \wedge c = idx \\
\quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \\
\quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\
\quad \sum(value) = \text{offset}\} & \\
.\{\text{idx}\}\text{recursive}.* \text{ where recursion } = \{x_1x_2\dots x_n| & offset > 0 \wedge c \neq idx \\
\quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \\
\quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\
\quad \sum(value) = \text{offset and } n = \text{c} - \text{idx}\} & \\
.\{\text{idx},\} & offset < 0 \wedge c > idx \\
.\{\text{c},\} & offset < 0 \wedge c = idx \\
.\{\text{idx},\} & offset < 0 \wedge c < idx \\
 & \quad \wedge abs(offset) < idx - c \\
 & \quad \wedge abs(offset) > 2 * (idx - c) \\
\Big|\Big|\Big(.\{\text{missingLength}\}recursion.*\Big) \text{ where recursion } = \{x_1x_2\dots x_n| & \text{otherwise} \\
\quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \\
\quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\
\quad \sum(value) = \text{offset and } n \neq \text{c} - \text{idx}\} & \\
\quad \text{and } missingLength = \text{idx} - n &
\end{cases}$$

$$[\![t.\texttt{offsetByCodePoints}(int\ idx, int\ offset) < c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases}
?! & c \leq 0 \\
.\{\text{idx},\} & offset = 0 \wedge c > idx \\
?! & (offset = 0 \wedge c \leq idx) \\
 & \quad \vee(offset > 0 \wedge idx \geq c) \\
.\{\text{idx}\}recursion.* \text{ where recursion } = \{x_1x_2\dots x_n| & offset > 0 \wedge idx < c \\
\quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \\
\quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\
\quad \sum(value) = \text{offset and } n < \text{c} - \text{idx}\} & \\
?! & offset < 0 \wedge c < idx \\
.\{\text{missingLength}\}recursion.* \text{ where recursion } = \{x_1x_2\dots x_n| & \text{otherwise} \\
\quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \\
\quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\
\quad \sum(value) = \text{offset and } n - \text{idx} < \text{c}\} \text{ and} & \\
\quad missingLength = \text{c} - n &
\end{cases}$$

$$[\![t.\texttt{offsetByCodePoints}(int\ idx{,}int\ offset) > c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases}
?! & offset = 0 \land c \geq idx \\
.\{\text{idx},\} & offset = 0 \land c < idx \\
.\{\text{idx}\}recursion.* \text{ where recursion } = \{x_1 x_2 \ldots x_n | & offset > 0 \land c \leq idx \\
\quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \\
\quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\
\quad \sum(value) = \text{offset}\} & \\
?! & offset > 0 \land c > idx \\
 & \quad \land c \geq idx + offset \\
.\{\text{idx}\}recursion.* \text{ where recursion } = \{x_1 x_2 \ldots x_n | & offset > 0 \land c > idx \\
\quad x = [\backslash\texttt{uD800} - \backslash\texttt{uDBFF}][\backslash\texttt{uDC00} - \backslash\texttt{uDFFF}] \text{ with value=2 or} & \quad \land c < idx + offset \\
\quad x = [\backslash\texttt{u0000} - \backslash\texttt{uD7FF}\backslash\texttt{uE000} - \backslash\texttt{uFFFF}] \text{ with value=1 and} & \\
\quad \sum(value) = \text{offset and } n > \text{c} - \text{idx}\} & \\
?! & offset < 0 \\
 & \quad \land c \geq idx + offset \\
.\{\text{idx},\} & \text{otherwise}
\end{cases}$$

$$[\![t.\texttt{regionMatches}(bool\ ignoreCase{,}int\ tO{,}String\ other{,}int\ oO{,}int\ len) {=}{=} c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases}
?! & c = true \land (tO < 0 \lor oO < 0 \\
 & \quad \lor len(other) < oO + len) \\
.* & (c = false \land (tO < 0 \lor oO < 0 \\
 & \quad \lor len(other) < oO + len)) \\
 & \quad \lor (len \leq 0 \land tO + len < 0) \\
.\{\text{tO} + \text{len},\} & len \leq 0 \land \neg(tO + len < 0) \\
.\{\text{tO}\}(?\texttt{i})\text{other}[\text{oO} : \text{oO} + \text{len}].* & ignoreCase = true \\
.\{\text{tO}\}\text{other}[\text{oO} : \text{oO} + \text{len}].* & otherwise
\end{cases}$$

$$[\![t.\texttt{startsWith}(String\ prefix{,}int\ tOffset) {=}{=} c]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases}
?! & c = true \land tOffset < 0 \\
.\{\text{tOffset}\}\text{prefix}.* & c = true \\
.* & c = false \land tOffset < 0 \\
?! & c = false \land len(prefix) = 0 \\
 & \quad \land toOffset = 0 \\
(.\{\texttt{0},\text{tOffset} - 1\}) & c = false \land len(prefix) = 0 \\
 & \quad \land tOffset > 0 \\
(.\{\texttt{0},\text{tOffset} - 1\})|(.\{\text{tOffset}\}((?!\text{prefix}).).*) & \text{otherwise}
\end{cases}$$

$$t.\texttt{toCharArray}()[i] {=}{=} c \rightarrow .\{\text{i}\}[\backslash\texttt{x}\{\text{c}\}].*$$

$$t.\texttt{toCharArray}()[i] \neq c \rightarrow .\{\text{i}\}[\hat{}\backslash\texttt{x}\{\text{c}\}].*$$

$$t.\texttt{toCharArray}()[i] < c \rightarrow .\{\text{i}\}[\backslash\texttt{x}\{\texttt{0}\} - \backslash\texttt{x}\{\text{c} - 1\}].*$$

$$t.\texttt{toCharArray}()[i] > c \rightarrow .\{\text{i}\}[\backslash\texttt{x}\{\text{c} + 1\} - \backslash\texttt{x}\{\text{MaxCodePoint}\}].*$$

A.4 Transformer mappers

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{concat}(String\ str)]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} \text{str}[0 : len(str) - len(regex)] & regex \text{ instance of String} \\ & \quad \wedge regex[len(regex) - len(str) : len(regex)] = str \\ NotSupported & \text{otherwise} \end{cases}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{repeat}(int\ count)]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} \text{regex} & count = 1 \\ ?! & count = 0 \wedge minLen(regex) > 0 \\ ?! & count \neq 0 \\ & \quad \wedge \neg(regex[0 : len(regex)/count]).repeat(count).matches(regex) \\ \text{regex}[0 : len(\text{regex})/\text{count}] & count \neq 0 \wedge regex \text{ instance of String} \\ & \quad \wedge (regex[0 : len(regex)/count]).repeat(count).matches(regex) \\ NotSupported & \text{otherwise} \end{cases}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{replace}(String\ oStr, String\ nStr)]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} replaceStrRec(\text{regex}, \text{oStr}, \text{nStr}, 0) & regex \text{ instance of String} \\ NotSupported & \text{otherwise} \end{cases}$$

$$replaceStrRec(reg, oStr, nStr, idx) \rightarrow$$

$$\begin{cases} (\text{nStr})|(\text{oStr})replaceStrRec(\text{reg}, \text{oStr}, \text{nStr}, idx + len(nStr)) & idx < len(regex) \\ & \quad \wedge str[idx : idx + len(nStr) = nStr \\ \text{str}[idx]replaceStrRec(\text{reg}, \text{oStr}, \text{nStr}, idx + 1) & idx < len(regex) \\ & \quad \wedge str[idx : idx + len(nStr) \neq nStr \\ () & \text{otherwise} \end{cases}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{replaceFirst}(String\ oStr, String\ nStr)]\!]_{\text{RE}} \rightarrow$$

$$\begin{cases} replaceStrRecFirst(\text{regex}, \text{oStr}, \text{nStr}, 0) & regex \text{ instance of String} \\ NotSupported & \text{otherwise} \end{cases}$$

$$replaceStrRecFirst(reg, oStr, nStr, idx) \rightarrow$$

$$\begin{cases} ((\text{oStr})\text{reg}[idx + len(\text{nStr}) : len(\text{reg})]) & idx < len(regex) \\ \quad |((\text{nStr})replaceStrRecFirst(\text{reg}, \text{oStr}, \text{nStr}, idx + 1)) & \quad \wedge str[idx : idx + len(nStr) = nStr \\ \text{str}[idx]replaceStrRecFirst(\text{reg}, \text{oStr}, \text{nStr}, idx + 1) & idx < len(regex) \\ & \quad \wedge str[idx : idx + len(nStr) \neq nStr \\ () & \text{otherwise} \end{cases}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{stripTrailing}(regex)]\!]_{\text{RE}} \rightarrow$$

regex((\x{9})|(\x{a})|(\x{b})|(\x{c})|(\x{d})|(\x{1c})|(\x{1d})|(\x{1e})|(\x{1f})
|(\x{20})|(\x{1680})|(\x{2000})|(\x{2001})|(\x{2002})|(\x{2003})|(\x{2004})
|(\x{2005})|(\x{2006})|(\x{2008})|(\x{2009})|(\x{200a})|(\x{2028})|(\x{2029})
|(\x{205f})|(\x{3000}))∗

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{substring}(int\ sIdx)]\!]_{\text{RE}} \rightarrow .\{\text{sIdx}\}\text{regex}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{substring}(int\ sIdx,int\ eIdx)]\!]_{\text{RE}} \rightarrow$$
$$\begin{cases} ?! & eIdx - sIdx < minLength(regex) \vee eIdx - sIdx > maxLength(regex) \\ .\{\text{sIdx}\}\text{regex}.* & \text{otherwise} \end{cases}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{toLowerCase}()]\!]_{\text{RE}} \rightarrow \sum_{i=0}^{len(\text{regex})-1} \left(set\right) \text{ where set} = \{x_1|x_2|\ldots|x_n|$$
$$x = lowerCase(y)\ and\ y = \{0, 1, \ldots FFFF\} \text{ and } \text{regex}[\text{i}].lowerCase() = x\}$$

$$regex \xrightarrow{\text{inject}} [\![t.\texttt{toUpperCase}()]\!]_{\text{RE}} \rightarrow \sum_{i=0}^{len(\text{regex})-1} \left(set\right) \text{ where set} = \{x_1|x_2|\ldots|x_n|$$
$$x = upperCase(y)\ and\ y = \{0, 1, \ldots FFFF\} \text{ and } \text{regex}[\text{i}].upperCase() = x\}$$

A.5 Results

Table A.1: failed tests

wlp	**negValue as char-points**
text.strip().charAt(1) != 'a'	\u10\u1c
text.strip().charAt(1) < 'a'	\u10\u1c
text.strip().charAt(1) <= 'a'	\u10\u1c
text.strip().charAt(1) == 'a'	\u1c\u61
text.strip().charAt(1) > 'a'	\u1c\u62
text.strip().charAt(1) >= 'a'	\u1c\u61
text.strip().codePointAt(1) != 1	\u10\u1c
text.strip().codePointAt(1) < 1	\u20\u1
text.strip().codePointAt(1) <= 1	\u20\u1
text.strip().codePointAt(1) == 1	\u20\u1
text.strip().codePointAt(1) > 1	\u10\u1c
text.strip().codePointAt(1) >= 1	\u10\u1c
text.strip().codePointBefore(1) != 1	\u1c
text.strip().codePointBefore(1) > 1	\u1c
text.strip().codePointBefore(1) >= 1	\u1c
text.strip().codePointCount(1, 1) != 1	\u1c
text.strip().codePointCount(1, 1) < 1	\u10
text.strip().codePointCount(1, 1) <= 1	\u10
text.strip().compareTo("a") != 1	\u1c\u62
text.strip().compareTo("a") < 1	\u1c\u62
text.strip().compareTo("a") <= 1	\u1c\u63
text.strip().compareTo("a") == 1	\u61\u1c
text.strip().compareTo("a") > 1	\u20\u61\u60\u20

wlp	negValue as char-points
text.strip().compareTo("a") >= 1	\u61\u1c
text.strip().compareToIgnoreCase("a") != 1	\u1c\u42
text.strip().compareToIgnoreCase("a") < 1	\u1c\u42
text.strip().compareToIgnoreCase("a") <= 1	\u1c\u43
text.strip().compareToIgnoreCase("a") == 1	\u41\u1c
text.strip().compareToIgnoreCase("a") > 1	\u205f\u41\u50
text.strip().compareToIgnoreCase("a") >= 1	\u41\u1c
text.strip().indexOf("a") != 1	\u20\u7b\u61
text.strip().indexOf("a") == 1	\u1c\u61
text.strip().indexOf("a") > 1	\u2029\u23\u61
text.strip().indexOf("a") >= 1	\u1c\u61
text.strip().indexOf("a", 1) <= 1	\u2008\u61\u3f\u61
text.strip().indexOf("a", 1) == 1	\u1c\u61
text.strip().indexOf("a", 1) > 1	\u20\u5f\u61
text.strip().indexOf("a", 1) >= 1	\u1c\u61
text.strip().indexOf(1) == 1	\u20\u1
text.strip().indexOf(1) > 1	\u20\u35\u1
text.strip().indexOf(1) >= 1	\u20\u1
text.strip().indexOf(1, 1) == 1	\u20\u1
text.strip().indexOf(1, 1) >= 1	\u20\u1
text.strip().lastIndexOf("a") == 1	\u1c\u61
text.strip().lastIndexOf("a") >= 1	\u1c\u61
text.strip().lastIndexOf("a", 1) != 1	\u20\u52\u61
text.strip().lastIndexOf("a", 1) < 1	\u2006\u66\u61
text.strip().lastIndexOf("a", 1) == 1	\u1c\u61
text.strip().lastIndexOf("a", 1) >= 1	\u1c\u61
text.strip().lastIndexOf(1) == 1	\u20\u1
text.strip().lastIndexOf(1) > 1	\u20\u67\u1
text.strip().lastIndexOf(1) >= 1	\u2001\u1
text.strip().lastIndexOf(1, 1) == 1	\u20\u1
text.strip().lastIndexOf(1, 1) >= 1	\u20\u1
text.strip().length() != 1	\u10\u1c
text.strip().length() == 1	\u1c
text.strip().length() > 1	\u10\u1c
text.strip().length() >= 1	\u1c
text.strip().offsetByCodePoints(1, 1) != 1	\u10\u1d
text.strip().offsetByCodePoints(1, 1) > 1	\u10\u1c
text.strip().offsetByCodePoints(1, 1) >= 1	\u10\u1c
text.strip().startsWith("a", 1)	\u1c\u61
text.stripLeading().charAt(1) != 'a'	\u1c\u11
text.stripLeading().charAt(1) < 'a'	\u1c\u11
text.stripLeading().charAt(1) <= 'a'	\u1c\u10
text.stripLeading().charAt(1) == 'a'	\u1c\u61
text.stripLeading().charAt(1) > 'a'	\u1c\u62
text.stripLeading().charAt(1) >= 'a'	\u1c\u61
text.stripLeading().codePointAt(1) != 1	\u1c\u10

wlp	**negValue as char-points**
text.stripLeading().codePointAt(1) < 1	\u20\u0
text.stripLeading().codePointAt(1) <= 1	\ud\u9\u20\u0
text.stripLeading().codePointAt(1) == 1	\u20\u1
text.stripLeading().codePointAt(1) > 1	\u1c\u10
text.stripLeading().codePointAt(1) >= 1	\u1c\u10
text.stripLeading().codePointBefore(1) != 1	\u1c
text.stripLeading().codePointBefore(1) > 1	\u1c
text.stripLeading().codePointBefore(1) >= 1	\u1c
text.stripLeading().codePointCount(1, 1) != 1	\u1c
text.stripLeading().codePointCount(1, 1) < 1	\u10
text.stripLeading().codePointCount(1, 1) <= 1	\u10
text.stripLeading().compareTo("a") != 1	\u1c\u62
text.stripLeading().compareTo("a") < 1	\u1c\u62
text.stripLeading().compareTo("a") <= 1	\u1c\u63
text.stripLeading().compareTo("a") > 1	\u2001
text.stripLeading().compareTo("a") >= 1	\u2001
text.stripLeading().compareToIgnoreCase("a") != 1	\u1c\u42
text.stripLeading().compareToIgnoreCase("a") < 1	\u1c\u42
text.stripLeading().compareToIgnoreCase("a") <= 1	\u1c\u44
text.stripLeading().compareToIgnoreCase("a") > 1	\u1680
text.stripLeading().indexOf("a") == 1	\u1c\u61
text.stripLeading().indexOf("a") > 1	\u2009\u6e\u61
text.stripLeading().indexOf("a") >= 1	\u1c\u61
text.stripLeading().indexOf("a", 1) <= 1	\u20\u61\u71\u61
text.stripLeading().indexOf("a", 1) == 1	\u1c\u61
text.stripLeading().indexOf("a", 1) > 1	\u20\u27\u61
text.stripLeading().indexOf("a", 1) >= 1	\u1c\u61
text.stripLeading().indexOf(1) == 1	\u20\u1
text.stripLeading().indexOf(1) > 1	\u1d\u2003\u1
text.stripLeading().indexOf(1) >= 1	\u2003\u1
text.stripLeading().indexOf(1, 1) == 1	\u20\u1
text.stripLeading().indexOf(1, 1) > 1	\u20\u56\u1
text.stripLeading().indexOf(1, 1) >= 1	\u20\u1
text.stripLeading().lastIndexOf("a") != 1	\u20\u61\u61
text.stripLeading().lastIndexOf("a") == 1	\u1c\u61
text.stripLeading().lastIndexOf("a") >= 1	\u1c\u61
text.stripLeading().lastIndexOf("a", 1) != 1	\u20\u76\u61
text.stripLeading().lastIndexOf("a", 1) < 1	\u2028\u78\u61
text.stripLeading().lastIndexOf("a", 1) == 1	\u1c\u61
text.stripLeading().lastIndexOf("a", 1) >= 1	\u1c\u61
text.stripLeading().lastIndexOf(1) == 1	\u20\u1
text.stripLeading().lastIndexOf(1) > 1	\u20\u27\u1
text.stripLeading().lastIndexOf(1) >= 1	\u20\u1
text.stripLeading().lastIndexOf(1, 1) == 1	\u20\u1
text.stripLeading().lastIndexOf(1, 1) >= 1	\u20\u1
text.stripLeading().length() != 1	\u1c\u10

wlp	**negValue as char-points**
text.stripLeading().length() == 1	\u1c
text.stripLeading().length() > 1	\u1c\u10
text.stripLeading().length() >= 1	\u1c
text.stripLeading().offsetByCodePoints(1, 1) != 1	\u1c\u12
text.stripLeading().offsetByCodePoints(1, 1) > 1	\u1c\u10
text.stripLeading().offsetByCodePoints(1, 1) >= 1	\u1c\u10
text.stripLeading().startsWith("a", 1)	\u1c\u61
text.stripTrailing().charAt(1) != 'a'	\u10\u1c
text.stripTrailing().charAt(1) < 'a'	\u10\u1d
text.stripTrailing().charAt(1) <= 'a'	\u10\u1c
text.stripTrailing().codePointAt(1) != 1	\u10\u1c
text.stripTrailing().codePointAt(1) > 1	\u10\u1c
text.stripTrailing().codePointAt(1) >= 1	\u10\u1c
text.stripTrailing().codePointBefore(1) != 1	\u1c
text.stripTrailing().codePointBefore(1) > 1	\u1c
text.stripTrailing().codePointBefore(1) >= 1	\u1c
text.stripTrailing().codePointCount(1, 1) != 1	\u1c
text.stripTrailing().codePointCount(1, 1) < 1	\u11
text.stripTrailing().codePointCount(1, 1) <= 1	\u10
text.stripTrailing().compareTo("a") != 1	\u61\u54\u20
text.stripTrailing().compareTo("a") == 1	\u61\u1c
text.stripTrailing().compareTo("a") > 1	\u61\u4d\u20
text.stripTrailing().compareTo("a") >= 1	\u61\u1c
text.stripTrailing().compareToIgnoreCase("a") != 1	\u41\u69\u20
text.stripTrailing().compareToIgnoreCase("a") == 1	\u41\u1c
text.stripTrailing().compareToIgnoreCase("a") > 1	\u41\u5b\u20
text.stripTrailing().compareToIgnoreCase("a") >= 1	\u41\u1c
text.stripTrailing().length() != 1	\u10\u1c
text.stripTrailing().length() == 1	\u1c
text.stripTrailing().length() > 1	\u10\u1c
text.stripTrailing().length() >= 1	\u1c
text.stripTrailing().offsetByCodePoints(1, 1) != 1	\u10\u1c
text.stripTrailing().offsetByCodePoints(1, 1) > 1	\u10\u1c
text.stripTrailing().offsetByCodePoints(1, 1) >= 1	\u10\u1c
text.trim().charAt(1) != 'a'	\u10\u10
text.trim().charAt(1) < 'a'	\u10\u10
text.trim().charAt(1) <= 'a'	\u10\u10
text.trim().charAt(1) == 'a'	\u10\u61
text.trim().charAt(1) > 'a'	\u10\u62
text.trim().charAt(1) >= 'a'	\u10\u61
text.trim().codePointAt(1) != 1	\u10\u10
text.trim().codePointAt(1) < 1	\u7b\u0
text.trim().codePointAt(1) <= 1	\u7b\u1
text.trim().codePointAt(1) == 1	\u7b\u1
text.trim().codePointAt(1) > 1	\u10\u12
text.trim().codePointAt(1) >= 1	\u10\u10

wlp	**negValue as char-points**
text.trim().codePointBefore(1) != 1	\u10
text.trim().codePointBefore(1) < 1	\u17\u1\u2\u0
text.trim().codePointBefore(1) <= 1	\u17\u1\u2\u1
text.trim().codePointBefore(1) == 1	\u1\u1b\u14\u1
text.trim().codePointBefore(1) > 1	\u10
text.trim().codePointBefore(1) >= 1	\u10
text.trim().codePointCount(1, 1) != 1	\u10
text.trim().codePointCount(1, 1) < 1	\u21
text.trim().codePointCount(1, 1) <= 1	\u10\u21
text.trim().compareTo("a") != 1	\u10\u62
text.trim().compareTo("a") < 1	\u10\u62
text.trim().compareTo("a") <= 1	\u10\u63
text.trim().compareTo("a") == 1	\u61\u10
text.trim().compareTo("a") > 1	\u61\u78\u20
text.trim().compareTo("a") >= 1	\u61\u10
text.trim().compareToIgnoreCase("a") != 1	\u10\u42
text.trim().compareToIgnoreCase("a") < 1	\u10\u42
text.trim().compareToIgnoreCase("a") <= 1	\u10\u43
text.trim().compareToIgnoreCase("a") == 1	\u41\u10
text.trim().compareToIgnoreCase("a") > 1	\u41\u53\u20
text.trim().compareToIgnoreCase("a") >= 1	\u41\u11
text.trim().indexOf("a") != 1	\u20\u30\u61
text.trim().indexOf("a") == 1	\u10\u61
text.trim().indexOf("a") > 1	\u12\u62\u61
text.trim().indexOf("a") >= 1	\u10\u61
text.trim().indexOf("a", 1) != 1	\u20\u3b\u61
text.trim().indexOf("a", 1) <= 1	\u20\u61\u20\u61
text.trim().indexOf("a", 1) == 1	\u10\u61
text.trim().indexOf("a", 1) > 1	\u20\u5f\u61
text.trim().indexOf("a", 1) >= 1	\u10\u61
text.trim().indexOf(1) == 1	\u29\u1
text.trim().indexOf(1) > 1	\u2d\u2d\u1\u1c
text.trim().indexOf(1) >= 1	\u7e\u1
text.trim().indexOf(1, 1) == 1	\u0017\u0001
text.trim().indexOf(1, 1) > 1	\ue\u13\u1
text.trim().indexOf(1, 1) >= 1	\u4c\u1\uc
text.trim().lastIndexOf("a") != 1	\u20\u71\u61
text.trim().lastIndexOf("a") == 1	\u10\u61
text.trim().lastIndexOf("a") > 1	\u020\u76\u61
text.trim().lastIndexOf("a") >= 1	\u10\u61
text.trim().lastIndexOf("a", 1) != 1	\u20\u52\u61
text.trim().lastIndexOf("a", 1) < 1	\u17\u40\u61
text.trim().lastIndexOf("a", 1) == 1	\u10\u61
text.trim().lastIndexOf("a", 1) >= 1	\u10\u61
text.trim().lastIndexOf(1) != 1	\u47\u1\u35\u1
text.trim().lastIndexOf(1) == 1	\u60\u1

wlp	negValue as char-points
text.trim().lastIndexOf(1) > 1	\u2e\u5c\u1
text.trim().lastIndexOf(1) >= 1	\u0031\u0001
text.trim().lastIndexOf(1, 1) == 1	\u2a\u1
text.trim().lastIndexOf(1, 1) >= 1	\u17\u1\u2\u7b\u1
text.trim().length() != 1	\u10\u21
text.trim().length() == 1	\u10
text.trim().length() > 1	\u10\u10
text.trim().length() >= 1	\u10
text.trim().offsetByCodePoints(1, 1) != 1	\u10\u10
text.trim().offsetByCodePoints(1, 1) > 1	\u10\u10
text.trim().offsetByCodePoints(1, 1) >= 1	\u10\u10
text.trim().startsWith("a", 1)	\u10\u61

Overview of Generative AI Tools Used

None

List of Figures

4.1 Overview of our approach 11
4.2 Syntax of parsers 15
4.3 Syntax of the WLP 17
4.4 Definition of Notation 20
4.5 Example of Mapper Notation 20
4.6 Definition of some Logic Method Mappers 21
4.7 Some Mappers for Transformation Methods 22

5.1 Evaluation pipeline 23

List of Tables

3.1 Examples of RegExes 8

4.1 Considered Java String Logic Methods 19
4.2 Considered Java String Transformation Methods 21

5.1 Classification of single input strings 27

6.1 Resulting performance metrics 30

A.1 failed tests 58

List of Algorithms

4.1 Overview of our approach 12
4.2 Extracting precondition 16

Bibliography

[AFH+19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[BAS+19] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, Santa Clara, CA, August 2019. USENIX Association.

[BvdMvL21] Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. Regular expressions with lookahead. *JUCS - Journal of Universal Computer Science*, 27(4):324–340, 2021.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[CT22] Nariyoshi Chida and Tachio Terauchi. On Lookaheads in Regular Expressions with Backreferences. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[CYLS07] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 317–329, New York, NY, USA, 2007. Association for Computing Machinery.

[Er88] M.C. Er. The weakest libera! preconditions of nondeterministic machines and robust programs. *Journal of Information and Optimization Sciences*, 9(3):415–420, 1988.

[GMZ20] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining input grammars from dynamic control flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 172–183, New York, NY, USA, 2020. Association for Computing Machinery.

[Gyö12] Révész György. *Introduction to formal languages György E. Révész.* Dover Publications, 2012.

[Hoa69a] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–580, 1969.

[Hoa69b] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[Hro11] Juraj Hromkovič. *Grammatiken und Chomsky-Hierarchie*, pages 348–407. Vieweg+Teubner, Wiesbaden, 2011.

[HZ16] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725, 2016.

[Java] Java language specification. `https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html`. Accessed: 2025-01-11.

[Javb] Java platform, standard edition & java development kitversion 23 api specification. https://docs.oracle.com/en/java/javase/23/docs/api/index.html.

[KSC+23] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models, 2023.

[LZ08] Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 83–93, New York, NY, USA, 2008. Association for Computing Machinery.

[MAK88] Robert N. Moll, Michael A. Arbib, and A. J. Kfoury. *An introduction to formal language theory.* Springer-Verlag, 1988.

[Ora] Oracle. The java tutorials - regular expressions. `https://docs.oracle.com/javase/tutorial/essential/regex/quant.html`. Accessed: 2024-12-02.

[Pet22] Alberto Pettorossi. *Automata theory and formal languages: Fundamental notions, theorems, and techniques.* Springer, 2022.

[PMP+15] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[Sch22] Michael Schröder. Grammar inference for ad hoc parsers. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2022, page 38–42, New York, NY, USA, 2022. Association for Computing Machinery.

[SGC23] Michael Schröder, Marc Goritschnig, and Jürgen Cito. An exploratory study of ad hoc parsers in python, 2023.