JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
**Daniel Simeon Jaburek**

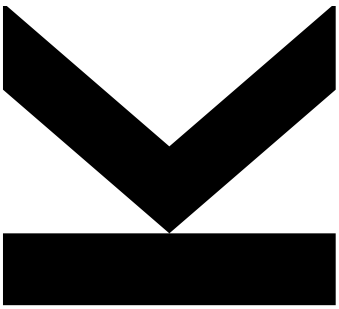
Submitted at
**Institut für Systemsoftware**

Supervisor
**DI Lukas Makor, Bsc.**

Co-Supervisor
**Dr. Christian Wirth**
**(Oracle Labs)**
**DI Josef Haider**
**(Oracle Labs)**

November 2022

# Support java.util.regex .Pattern in TRegex for increased performance



Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Computer Science

**JOHANNES KEPLER**
**UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

## Statutory declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references. This printed thesis is identical with the electronic version submitted.

Wels, 10.11.2022
Place, Date


Signature

# Zusammenfassung

Reguläre Ausdrücke werden häufig zum Abgleichen eines Musters in einer Datenbank, einem Text, einer Datei usw. verwendet. Da sie flexibel und beliebig komplex sind, wurden viele verschiedene Engines für reguläre Ausdrücke entwickelt. Eine dieser Engines ist die leistungsstarke Java-Engine für reguläre Ausdrücke, die weltweit auf Milliarden Geräten verwendet wird.

Das Truffle Language Implementation Framework (Truffle) arbeitet auf der Grundlage der Verwendung selbstmodifizierender abstrakter Syntaxbäume, die dazu beitragen, die Effizienz von Programmiersprachen zu verbessern. Truffle wird mit dem Graal-Compiler verwendet, der einige hohe Standards für eine Leistungssteigerung setzt. Truffle stellt auch seine eigene Engine für reguläre Ausdrücke namens TRegex, eine automatenbasierte ECMAScript-kompatible Engine für reguläre Ausdrücke.

Das Ziel dieser Bachelorarbeit ist es, den Zugang zur schnelleren Regular Expression Engine von Truffle von Java aus zu öffnen und deren Performance-Steigerung zu verifizieren. Daher umfasst es die Erstellung eines neuen Parsers, der reguläre Java-Ausdrücke in Ausdrücke transpiliert, die TRegex interpretieren kann, sowie alle damit verbundenen Aspekte. Einige von ihnen umfassen Forschung und Leistungsmessung. Dieser Prozess bestätigt die Leistungssteigerung und ermöglicht eine Beschleunigung um den Faktor 1,7 für Benutzer, die den "neuen" Parser verwenden, der den Zugriff auf die TRegex-Engine von Java-Anwendungen aus ermöglicht.

# Abstract

Regular expressions are commonly used for matching a pattern in a database, text, file and so on. As they are flexible and arbitrary complex a lot of different regular expression engines have been developed. One of such engines is the powerful Java regular expression engine which is used on billion devices worldwide.

The Truffle language implementation framework (Truffle) works on the base of using self-modifying Abstract Syntax Trees which helps improving the efficiency of programming languages. Truffle is used with the Graal compiler which sets some high standards for a performance increase. Truffle provides its own regular expression engine called TRegex, an automata-based ECMAScript compliant regular expression engine.

The objective of this thesis is to open the access to the faster regular expression engine of Truffle from Java and to verify its performance increase. Thus it covers creating a new parser which transpiles Java regular expressions into expressions TRegex can interpret and all aspects connected to it. Some of them include research and measuring the performance. This process results in confirming the performance increase and enabling a speedup of the factor 1,7 for users using the "new" parser which enables access to the TRegex engine from Java applications.

# 1 Introduction

*Truffle Regular Expressions*, also called *TRegex* henceforth, is a regular expression engine (see Section 2.4). It is implemented in Java on top of *Truffle* (see Section 2.3) to help optimize its execution. Due to Truffle, and the way it is implemented, it should increase the performance of regular expressions, also called *regex*. The engine is part of the *GraalVM* repository (see Section 2.2).

TRegex was originally designed for *ECMAScript* (JavaScript) [2] and supports all its current regular expression features. Additionally, support for Ruby and Python has been added. As Ruby and Python have different regular expression syntaxes a transpiler system has been set up.

The regular expression engine used by *OpenJDK* [7], also called *Java Regex* henceforth, itself is a backtracker which explores all possible options of how to parse a text by a regular expression. This can take up time and lead to a exponential run time in the worst case. On the other hand, TRegex can solve this in linear run time which is indeed faster in almost all cases especially if the input data and the regular expression become longer and more complicated.

The goal of this thesis is to enable access to the faster regex engine (TRegex) for Java workloads. Users should be able to benefit from the enhanced performance from a plain Java application by using the `java.util.regex.Pattern` interfaces. In order to achieve this goal the tasks of the project are described in the following paragraphs.

Enabling Java applications to utilize TRegex is one of the main objectives. We reach that goal by implementing a bridge between the interface `java.util.regex.Pattern` and TRegex. In other words, it must be possible to parse a regex via TRegex within a Java Application and retrieve all the information that would be available when parsing that regex with the Java Regex engine.

Being able to not just access TRegex from a Java application but also to provide support for Java regular expressions, which differ in certain parts compared to the regular expressions of JavaScript, a transpiler needs to be implemented. One of the most challenging tasks of this assignment has been to identify all differences between the two regular expression languages and how to translate Java's regex into a language TRegex can interpret and compile.

Thus, the differences needed to be researched and documented. Based on those findings, a translation step was implemented to be able to forward the regular expressions from Java Regex to TRegex. A translation between the two regular expression languages is not enough to work properly. In addition to that, it must be thought of how to communicate and work with the TRegex engine. Even though Ruby and Python have already been added to the TRegex system with their transpiler and parser, it is not just copy and paste as Java does differ from those two languages. Therefore, a combination of those two templates and additional coding will be necessary to enable access from Java Regex to TRegex. The parsers for Ruby and Python have been implemented a short while ago. Thus, a new principle can be developed of how to improve the layout and structure of the new parser to set an example for the other two parsers.

For a full understanding of what features and functionalities are supported and what might still cause problems during the implementation a full documentation of the process is created. Furthermore, all covered features and various not yet covered features are documented. This will help future developers adding onto this project and completing the support for Java's regular expression API functionality.

After implementing the transpiler the effectiveness and fullness is tested which requires a testing suite. The testing cases are set up before and while implementing this project to help programming to the user's need. Therefore, test cases need to be created to cover all necessary features. The testing also helps determine the coverage of Java Regex in TRegex. This helps listing features which can be supported

and others which are currently not supported. After testing and verifying which features are supported, benchmarks are set up and run to evaluate the efficiency of the transpiler. Those results then answer the questions: Is TRegex faster than Java Regex? Where are potential issues?

Lastly, after successful implementation and testing as well as benchmarking the transpiler, this project is added to the GraalVM repository to provide access to its features to all users.

As mentioned before this thesis will verify whether a regular expression expressed in the Java Regex format can be more efficiently parsed with TRegex. If so it will solve time problems as the performance increases, which enables it to parse large files or databases to match a pattern and achieve many other use cases.

# 2 Fundamentals

In this chapter, we introduce a few fundamental topics relevant for the rest of this thesis. The first Section 2.1 presents some basics starting with regular expressions. Section 2.2 will talk about GraalVM and its usage in this project. Afterwards, Section 2.3 will give an overview of the open source library Truffle. Lastly, Section 2.4 brings the concept of TRegex closer to the reader.

## 2.1 Regular Expression

"Regular expressions are patterns used to match character combinations in strings" [9]. A regular expression can be referred to as a search pattern in a text. Its main purpose is to "find" or to "find and replace" a pattern in a database, text, file and so on. A regex consists of one or more literals, operators and other characters.

A regular expression is normally passed into a processor which does some small adjustments like changing escape sequences according to its programming language. Afterwards, the compiler forwards the regular expression to the Regex Engine which generates different outputs for a corresponding input. The output can be all matches in a set, or just a `true`/`false` if a match is found and so on. Normally it returns an instance of regex result or matcher class. This scheme is briefly illustrated in Figure 2.1. On top a sample regular expression is presented.

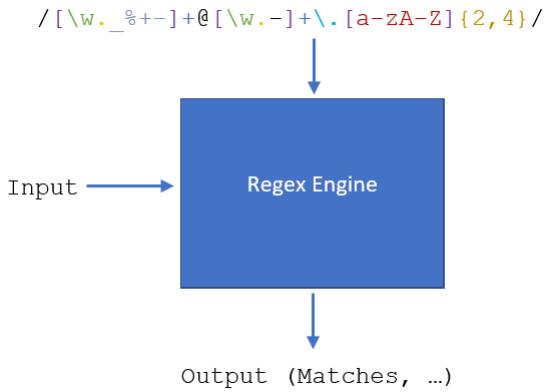



**Figure 2.1:** Concept of Regular Expression Engine

The format of a regular expression is as follows:

"/*pattern_to_match*/*flags*"

The term `pattern_to_match` is a combination of different characters consisting of different tokens:

- Simple patterns which check for an exact sequence.
  E.g.: `/abc/` will match `abc`
- Special characters:
  - Assertions
  - Character Classes
  - Groups
  - Quantifiers
  - Unicode property escapes
  - ...

For more information on how to use regular expression and how they work can be found online under https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions.

### 2.1.1 Java Regular Expressions

In Java, regular expressions and their accompanying classes are often referred to as Java Regex. In order to be able to use Java Regex its corresponding package as listed in Listing 1 needs to be imported.

```
import java.util.regex.*;
```

**Listing 1:** Importing Java Regex

The package includes the following classes:

- **`Pattern Class`**
  The Java API doc states: "A compiled representation of a regular expression" [8]. Therefore, in order to use regular expressions with Java Regex those expressions need to be compiled into an object of this class.
- **`Matcher Class`**
  This class is "an engine that performs match operations on a character sequence by interpreting a Pattern." [6] An object of this class can be created by calling the `p.matcher(str);` - function where `str` represents a string on which a pattern `p` is matched.
- **`MatchResult` Interface**
  This interface provides query methods to determine the results of a pattern matching operation. `MatchResult` contains several information on the result after parsing a string as the matched substring, the starting and ending positions of the match, the *groups* matched and so on.

Listing 2 gives an example how those classes can be used to perform pattern matching on a character sequence. The first line shows how to parse a regular expression and store the result in a `Pattern` object. The next line creates a `Matcher` object which the pattern and a corresponding text to check is passed to. In the last line a `boolean` variable, storing whether a match was found, is created.

```
Pattern p = Pattern.compile("ab+");
Matcher m = p.matcher("abbb");
boolean b = m.matches(); // b is true in this case
```

**Listing 2:** Simple pattern matching

The lines shown in Listing 2 can also be shortened to one line as presented in Listing 3.

```
boolean b = Pattern.matches("ab+", "abbb"); // b is true as well
```

**Listing 3:** Simple pattern matching (short)

Listings 2 and 3 give very basic examples. Of course, the API is much richer and provides support for many more use cases.

## 2.2 GraalVM

GraalVM[1] is a Java virtual machine and JDK implemented in Java. Its main part is the *GraalVM Compiler*, a Java compiler written in Java which supports all Java-bytecode-based languages like Scala, Kotlin, and of course, Java itself. The second part is the *GraalVM Native Image* for ahead-of-time-compilation of Java applications. The third part adds support for many more programming languages based on Truffle, like Python, JavaScript, Ruby, LLVM and many more. As its Github page states "GraalVM is a high-performance JDK distribution designed to accelerate the execution of applications written in Java and other JVM languages along with support for JavaScript, Ruby, Python, and a number of other popular languages." [3].

## 2.3 Truffle Language Implementation Framework

The Truffle language implementation framework[2] or henceforth *Truffle* is an open source library to implement programming languages. It is designed to run existing languages like JavaScript, Python, or

[1] https://www.graalvm.org/
[2] https://www.graalvm.org/22.0/graalvm-as-a-platform/language-implementation-framework/

C, but is also a good fit to build a new programming language from scratch. It works on the base of using self-modifying Abstract Syntax Trees, short AST. In Figure 2.2 we see how Truffle is integrated in GraalVM to provide the support it does.

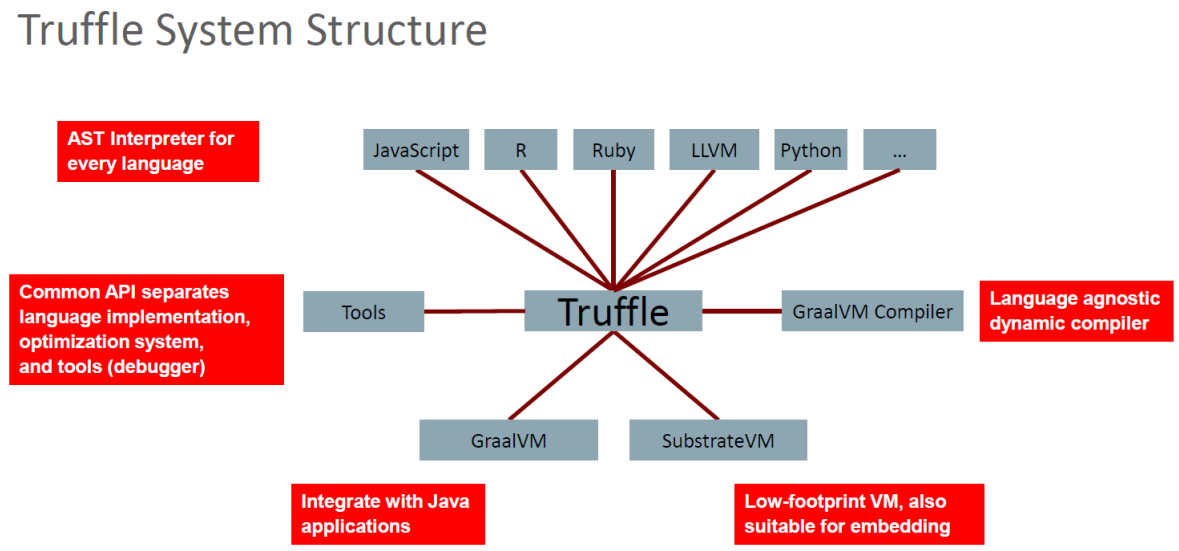



**Figure 2.2:** Truffle System Structure [1]

A big feature of Truffle is that it allows one to implement an own programming language and run it efficiently in GraalVM. It simplifies the implementation by automatically optimizing and generating high-performance code through interpreters [11].

## 2.4 Truffle Regular Expressions (TRegex)

According to its documentation, the Java Regex Pattern-Matching Engine is a backtracking non-deterministic finite automaton (NFA)-based engine [8]. Deterministic algorithms will always, for a particular input, produce the same output going through the possible states compared to the non-deterministic algorithms which, for the same input, may produce different outputs in different runs (Figure 2.3). Therefore the engine tries to go as far down a path until there is no further way to continue so it backtracks to another possible path it can take until it finds a matching path.

Figure 2.3 illustrates on the left side if the algorithm has reached the state **2** and additionally if the next character input would be `c` that State **2** does not have an outgoing pointer with `c` so it backtracks to the state **1** and tries the other path and goes to state **3** where an outgoing arrow with `c` does exist. This whole process can result into calculating and trying a lot of alternatives till finding a solution.

Determinism, which TRegex is based on, on the other hand has the advantage to provide only one specific path for each character presented. This leads to producing always the same output and going the same path which can be faster and lead to determination of all matches in linear time. The disadvantage is that creating such an automaton is more complex and may not always be possible.

Josef Haider describes the concept of TRegex for all regular expressions in his thesis [5] as follows:

- Parse the regular expression
- Create a non-deterministic finite automaton (NFA) from the parser tree
- Create a deterministic finite automaton (DFA) from the NFA

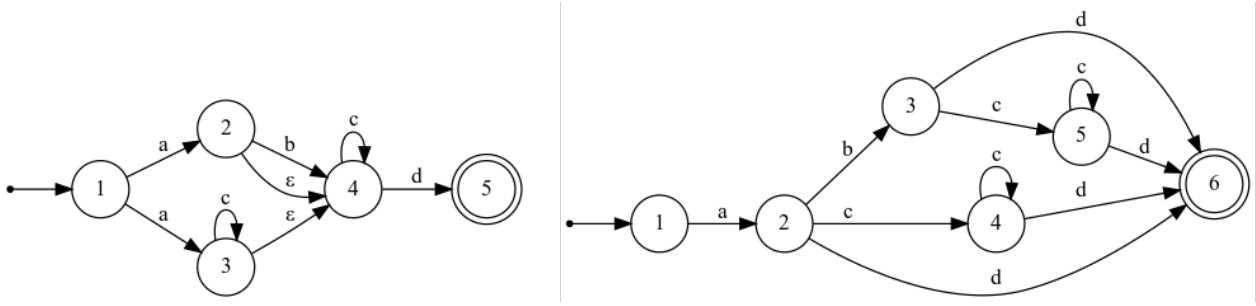



**Figure 2.3:** Non-determinism vs Determinism [10]

- Compile the resulting DFA to a regex matcher with the Graal compiler
- If the conversion to DFA fails for any reason, it bails out and the hand the expression to a different fall-back engine

As stated above TRegex has to look at each character once while Java Regex might even look at each character more than a thousand times. In addition to that TRegex can compile regexes directly which Java cannot do. Java Regex simply said has a loop where it looks what it can do and tries all possibilities. It also checks for each index position whereas TRegex reads the letter of the string checks if it can match it with the DFA and continues on. One can speak of a "unrolled" loop. Thus, the main advantages of TRegex: a deterministic automaton with linear time in comparison to the backtracker & TRegex is JIT compilable. Those two combined lead to a huge performance increase.

# 3 Using TRegex for Java regular expressions

In this chapter, we will present the concepts behind using TRegex for Java regular expressions. Section 3.1 gives an overview of the task, while Section 3.2 will go into more details around the differences of regular expressions in Java compared to JavaScript.

## 3.1 Overview of the task

As stated in the introduction the task of this thesis is to enable Java applications to utilize TRegex. The *Java regular expression API* [8] is a complex and powerful tool for pattern matching but it has its limits. Thus, the aim is to allow the parsing of regluar expressions to be performed by TRegex. Furthermore, after achieving this its advantages and disadvantages compared to Java's own engine are evaluated.

## 3.2 Differences of Regular Expressions in Java compared to JavaScript (ECMAScript)

Regular expressions are not a universal standard - each programming language has its own set of syntaxes and semantics. TRegex supports regular expressions as specified by the ECMAScript specification [2]. In this work, we want to use it to execute regular expressions as specified by the Java language specification [8]. In this section, we compare the syntactic and semantic differences between ECMAScript's and Java's regular expressions.

In order to have full support for Java's regular expression syntax, we need to support all those features with JavaScript's regular expression engine. Where JavaScript lacks a certain feature relevant for Java, this needs to be mediated somehow. Features supported by JavaScript but not by Java pose less of a problem.

Note however, that they are of *some* concern still. Features available in JavaScript, but not in Java, might trigger errors in Java's regex engine. If JavaScript ignores this, it will do "something" with the input, and not throw an error, e.g.: `("invalid Regexp")`, as maybe expected.

The following bullet points are the main categories wherein Java and JavaScript differ.

- **Nested Character Class**
  Java supports nested character classes like `[ab[cd]ef]` where an opening square bracket inside the class begins a nested character class. JavaScript interprets "`[`" and "`]`" as a literal character and adds them to the character class.
- **Character Class Intersection**
  Java supports intersecting two character classes with the following syntax: `[base&&[intersect]]` or `[base&&intersect]`. For example: `[a-z&&[$^$aeiuo]]` matches a single letter that is not a vowel.
- **POSIX class**
  Java supports the use of POSIX classes (predefined character classes) like e.g.: `\p{Digit}` which matches any single digits. POSIX classes can also be used inside character classes.
- **Escapes**
  Java supports some character escapes, octal escapes, ... which JavaScript does not support:
  - Escape sequence - `\Q...\E`
  - Character escape
    `\a` (match "alert" or "bell" control character)
    `\e` (match "escape" control character)

  - Octal escape
    `\0100 - \0177` match at the specified position in ASCII table
    `\0200 - \0377` match at the specified position in active code page

  Almost all of the escape characters do also work inside a character class.

- **Shorthand Character Classes**
  Several shorthand character classes are supported equally by Java and JavaScript. There are two though which are supported by Java only:

  - `\v` which matches any single vertical whitespace character.
  - `\h` which matches any single horizontal whitespace character.

  Shorthands like `\s` are translated into different character classes internally in Java and JavaScript and need to be therefore translated beforehand. For example `\s` equals `[\f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000 \ufeff]` in JavaScript and `[\t\n\x0B\f\r]` in Java.

- **Anchor**
  In Java the following anchors have been added compared to JavaScript:

  - `\A` which matches at the start of the string the regex pattern is applied to.
  - `\G` which matches at the end of the previous match during the second and following match attempts.
  - `\z` which matches at the end of the string.
  - `\Z` which matches at the end of the string as well as before the final line break (if any).
  - `\$` works almost the same in Java as in JavaScript with the difference that it also matches before the final line break in the string (if any) in addition to matching at the very end of the string.

- **Word Boundaries**
  Word Boundaries match in specific circumstances as stated in the following:

  - `\b` matches at a position that is either followed by a word character but not preceded by one, or on the other hand that is preceded by word character but not followed by one.
  - `\B` matches at a position that is either followed and preceded by a word character or that is not preceded and followed by a word character.

  In Java and JavaScript they are the same besides Java also supporting the boundaries in Unicode mode whereas JavaScript just supports the ASCII-boundaries.

- **Possessive Quantifier**
  There are three types of quantifiers: greedy, lazy and possessive. Last one is only supported by Java. The different possessive quantifiers are: `"?+"`, `"*+"`, `"++"`, `"\{n,m\}+"` and `"a\{2,\}+a"`.

- **Unicode Characters and Properties**

  - `\x{FFFF}` matches 1 to 4 hexadecimal digits to a specific Unicode code point.
  - Unicode category
    In Java many Unicode categories exists which are pretty much just an abbreviation of POSIX classes.

  * `\p{...}` matches a single Unicode code point that has the specified property.
  * `\P{...}` matches a single Unicode code point that does not have the specified property.

  For a lot of those POSIX classes also an corresponding `\p{Is...}` or `\p{In...}` exist which denotes the same Unicode category.

- **Atomic Groups**
  Java supports atomic groups which prevent the regex engine from backtracking back into the group after a match has been found for the group. For example: `a(?>bc|b)c` matches `abcc` but not `abc`.

- **Mode Modifier**
  The *mode* modifier syntax in Java consists of parentheses and a question mark. Depending where it is positioned it can affect the whole regular expression or just part of it. The different types are:
  - `(?letters)` at the start of the regex would affect the whole regex and overrides any options set outside the regex. For example `(?i)a` would activate the case insensitive mode and thus match `a` and `A`.
  - `(?letters)` in the middle of the regex would only affect the parts to the right of the modifier. If used in a group, it only affects part of the regex inside that group to the right of the modifier. For example `te(?i)st` would match `test` and `teST` but not `TEst` or `TEST`.
  - `(?letters:regex)` is a non-capturing group with modifiers that affects only the part of the regex inside the group (`regex`).
  - `(?on-off)` and `(?on-off:regex)` can turn on or off the set modifier.

  The *mode* modifier is not supported by JavaScript.

These information were gathered primarily with the help of [4]. In Chapter 4, we will show solutions how to provide support for those differences in the TRegex engine.

# 4 Implementation

In this chapter we will present the relevant changes implemented for this work. In Section 4.1 an overview of the structure of the implementation and its accompanying files is given. Section 4.2 goes into more depth on how to create the parser. We have shown in the previous chapter that there is some difference between Java's and JavaScript's regex. Therefore, in this chapter we will show how a transpiler can solve that. Even though Java and JavaScript and the connecting regular expression engines are quite similar there are still dissimilarities to be implemented for striving to get as close to full coverage of Java Regex as possible. Therefore, Section 4.3 talks about how those differences have been implemented. Lastly, challenges, which occurred while implementing the parser, are presented in Section 4.4.

Similar transpilers have been programmed in the past for the languages Ruby and Python to enable their execution with TRegex. To avoid making syntactics and algorithms anew the structure and functions implemented for the other languages were taken as a template to lean on. As a parser for Ruby and Python has already been implemented it provides a prototype of how to implement the same for Java. Also, the `RegexLexer` and `JSRegexParser` classes are noteworthy as they are responsible for parsing regular expressions and building the corresponding AST objects in JavaScript.

The parser for Python is using the concept of parsing the "regex string" and adapting it to fit to ECMAScript-Standards. Python's parser class provides methods to alter the regular expression string and returns the fitting string back to the regular expression engine. On the contrary, Ruby's parser has been adjusted to use the AST-Builder to fill the abstract syntax tree which will be passed to the engine later on. Figure 4.1 shows the syntax tree for the regular expression `a(b|c)`. The top (5) signalises a group `0` which contains a concatenation of "`a`", the group `1` and the end marker "`\$`". Group `1` consists of the character class `b-c`. The picture on the right in Figure 4.1 is a representation of the regular expression `a(b|.*c)`. "`.`" is translated into the last long line which are Unicode codes for every single character falling into this group.

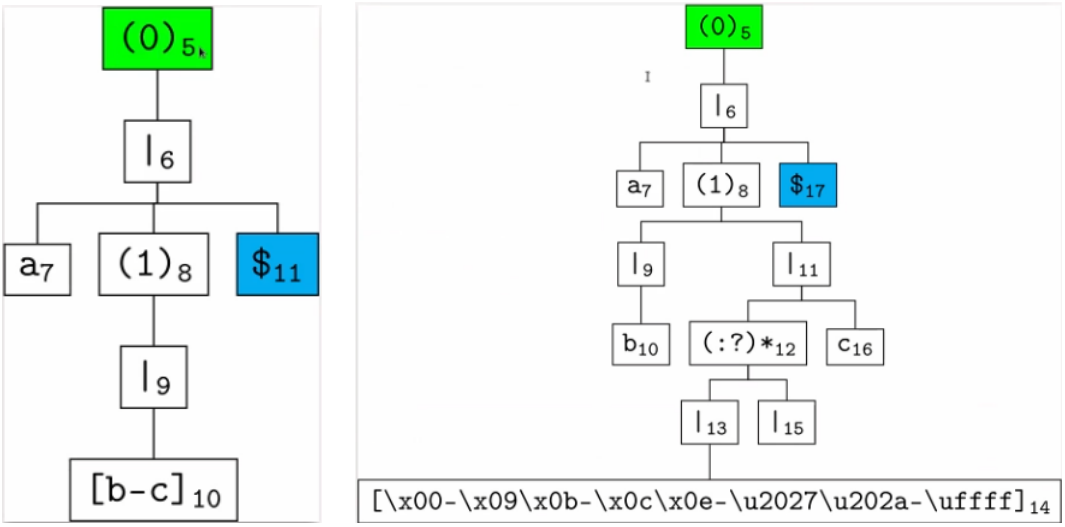



**Figure 4.1:** Abstract Syntax Trees

In the beginning of the construction of the Java Parser the template of Python has been used as a model but later on switched to Ruby's one. A major reason for this change was that Oracle Labs was planning on adjusting the Python Parser to the same scheme as the one Ruby has been adapted to. But not only `RubyRegexParser` has been used to provide a base, also `JSRegexParser` and `RegexLexer` have been used to help implement the requested features. Why writing new code if you can use already existing one and adjust it to the needs required?

Towards the end of the project the concept of `RegexLexer` and `JSRegexParser` has been applied to the Java Regex Parser. This meant splitting up the file `JavaRegexParser` into two files: `JavaLexer` and `JavaRegexParser`. The lexer is responsible for creating different tokens which will be talked about later in this section. On the other hand, the parser takes those tokens and creates an abstract syntax tree to pass on to TRegex.

## 4.1 Structure

The parser for Java to JavaScript Regex is split up in a few separate class files:

- `JavaFlavor.java`
  An implementation of the `java.util.Pattern` regex flavor.
- `JavaFlags.java`
  An immutable representation of a set of `java.util.Pattern` regular expression flags.
- `JavaRegexParser.java`
  This class receives tokens of `JavaLexer.java` and creates an abstract syntax tree.
- `JavaLexer.java`
  This file creates the tokens for `JavaRegexParser.java` by parsing the regular expression.

All the files above are stored in `com.oracle.truffle.regex.tregex.parser.flavors.java` package. `JavaRegexParser.java` and `JavaLexer.java` implement together the parsing and translating of `java.util.Pattern` regular expressions to ECMAScript regular expressions.

### 4.1.1 JavaFlavor.java

The flavor class is extended by `RegexFlavor` class which is an implementation of a dialect (= flavor) of regular expressions other than JavaScript. It provides support for validating and parsing of regular expressions. It consists of a constructor and two `create`-methods to create a parser and a validator.

### 4.1.2 JavaFlags.java

This class represents all the flags Java Regex could set, the flags "`idmsuxU`". One can add or remove a flag to the flags set in the expression (necessary for enabling mode modifiers as described in Section 3.2). It also adds all `is...`-methods to check if a certain flag is set or not.

### 4.1.3 BaseLexer.java

When creating the `JavaLexer`, the `RegexLexer` has been looked at to get an idea what needs to be included in a lexer. In addition, while developing the class `JavaLexer` many similarities between itself and `RegexLexer` became apparent which opened the opportunity to clean up and simplify the code by creating a base class of which both, the `JavaLexer` and the `RegexLexer`, classes then derived from.

The `BaseLexer` class offers basic functionality for parsing a regular expression character by character and transforming its syntax into tokens that can be forwarded to the parser.

### 4.1.4 JavaRegexParser.java

This class consists of the actual Java Regex Parser to translate regex, which is Java compliant, to regex, which TRegex can interpret. It contains several functions, internal classes and structures.

This class, as already stated above, receives tokens from the lexer and interprets those to create an abstract syntax tree. It creates a syntax tree which TRegex can interpret and run faster than Java would do it. Its functions and concept will be described in the next few paragraphs.

#### 4.1.5 JavaLexer.java

This class extends the class `BaseLexer.java` and extends functions needed only for Java to be able to interpret Java Regex correctly to be able to translate it into a JavaScript-compliant regular expressions.

### 4.2 Parser

A parser is created by calling the `createParser()`-method as listed in Listing 4. The first parameter `language` is necessary for the `RegexParserGlobals` reference for the `JSRegexParser` class. The `source` contains the regular expression to parse and the buffer variable is as its name states a buffer. A `JavaRegexParser` object is created by passing on the `source` and a new AST-builder.

```
public static RegexParser createParser(RegexLanguage language,
    ↪ RegexSource source, CompilationBuffer compilationBuffer) throws
    ↪ RegexSyntaxException {
     return new JavaRegexParser(source, new RegexASTBuilder(language,
        ↪ source, makeTRegexFlags(false), compilationBuffer));
}
```

**Listing 4:** createParser()-method

In the constructor of `JavaRegexParser` several attributes are set to be able to smoothly parse the regular expression. Some of the attributes are the `inFlags` (flags of the regex), `astbuilder` (a builder which sets up a abstract syntax tree like in Figure 4.1) and the `lexer` which is created in the constructor.

#### 4.2.1 Parsing the Regex

After the parser has successfully been created, the regex is parsed. For that, the `parse()` method as shown in Listing 5 is executed. For a better overview of the extent of this function the bits and pieces inside the cases of the `switch-case` have been removed. The parser can differentiate between the different kinds of tokens and perform certain actions necessary for building the correct regular expression the TRegex engine can understand. This method includes a loop which iterates through the whole regex until we have reached the end: `while (lexer.hasNext()){...}`. Inside the loop we receive a token and split up the actions according to its kind.

```
public RegexAST parse() {
    astBuilder.pushRootGroup();
    Token token;
    boolean openInlineFlag = true;
    while (lexer.hasNext()) {
        token = lexer.next();
        switch (token.kind) {
            case caret:
            case dollar:
            case wordBoundary:
            case nonWordBoundary:
            case backReference:
            case quantifier:
```

```
            case anchor:
            case alternation:
            case inlineFlag:
            case captureGroupBegin:
            case nonCaptureGroupBegin:
            case lookAheadAssertionBegin:
            case lookBehindAssertionBegin:
            case groupEnd:
            case charClass:
            case comment:
        }
    }
    if (!astBuilder.curGroupIsRoot()) {
        throw syntaxErrorHere(ErrorMessages.UNTERMINATED_GROUP);
    }

    return astBuilder.popRootGroup();
}
```

**Listing 5:** parse()-method

To be able to determine into which point a term falls into we need to parse the regex further. This will not be done by the parser but by the lexer with its method `getNext()` which is called inside the method `next()` which the parser calls on. `getNext()` is also set up with a `switch-case` where the specific syntax characters are differentiated on. Connected to each character a certain method is called upon and later on the corresponding token is returned to the parser.

This pattern described so far is a pattern for a general parser (see `JSRegexParser` and `RegexLexer`) and not a focus in this work. For more details and example for a parser structure take a look at https://github.com/oracle/graal/tree/master/regex/src/com.oracle.truffle.regex/src/com/oracle/truffle/regex/tregex/parser. Furthermore, we want to focus on how the differences have been put into action.

## 4.3 Transpiler

In this subsection the implementation of the unique features of Java Regex are presented. It will show how the differences as described in Section 3.2 are implemented in the transpiler itself.

### 4.3.1 Token

A token is an object which can be passed from two different classes and inform the other which regular expression part has been traversed. It includes its `kind` and any other additional necessary information. For example a character class can be created as a token's inner class `CharacterClass` and given the information of the elements inside and so on. This token then is passed on to the parser which can extract all the information it needs to fill its abstract syntax tree.

Furthermore, as already stated multiple times, the lexer returns the different tokens with all the information needed to the parser which differentiates their kinds to perform additional actions.

### 4.3.2 Character Class

Character classes are a bit different to those of JavaScript and in order to transpile them into character classes TRegex can comprehend we use a similar approach Ruby is using. The function `characterClass()`, as listed in Listing 6, creates the token for a character class with its `CodePointSet`. For that it collects

all the information needed and fills the `CodePointSetAccumulator curCharClass` which is a set full of all the different characters the character class shall consist of.

```
private Token characterClass() {
    curCharClassClear();
    collectCharClass();
    return Token.createCharClass(curCharClass.toCodePointSet(),
       ↪ curCharClass.matchesSingleChar());
}
```

**Listing 6:** characterClass()-method

Filling the character class we need to distinguish if a unicode category is inside the class. For remembrance: a unicode category can look like `[\p{ASCII}]` which matches all ASCII letters. A unicode category can also exist outside the character class and is looked at again in the paragraph POSIX class 4.3.3. With several functions like `unicodeParseProperty()` and some others it is made possible to extract all the codepoints of the category and add them to the character class. A note to point out is that with a capital "`P`" this category has to be treated negated.

Another aspect of Java's character classes are the nested character classes and in addition to that the intersection of different classes. The principle behind nested character classes is to call the `collectCharClass()` serveral times recursively depending how deeply nested the classes are and adding the codepoints to the accumulator. If the class is intersecting with another class we collect the intersecting class codepoints and add them correctly into the `curCharClass`-accumulator.

In case of having the flag set Ignore Case we will have to create the fully foldable characters set and in the end close the case. Closing the case just means to add the case-folded version of the characters inside the accumulator to have a combined version of it. Doing so we have taken the advantage of some methods of Ruby's code and their base concept.

### 4.3.3 POSIX class

POSIX classes are concepts similar to predefined character classes or Unicode categories. They simply translate a term into a character class of its valid characters. While implementing this feature one has to note that for different categories, scripts, and so on there are corresponding `\p{Is...}` or `\p{In...}` and also `\p{...}` and its counterpart `\P{...}`.

We therefore need to detect a small "`p`" or capital "`P`" as well as if an "`Is`" or "`In`" is at the beginning of the Unicode property. After trimming the property to just its main part (without any "`Is`" or "`In`", for example "`Letter` instead of "`IsLetter`") we differentiate the property between a *General Category*, a *Script*, a other supported property or bail out if it is not supported. Then the property is trimmed to the encoding and we can return the property as codepoint set as listed in Listing 7. All predefined codepoint sets are stored in a seperate file and are returned as a set to the parser.

```
if (UNICODE_POSIX_CHAR_CLASSES.containsKey(propertySpec)) {
    property = getUnicodePosixCharClass(propertySpec.toLowerCase());
} else if (UnicodeProperties.isSupportedGeneralCategory(propertySpec,
   ↪ true)) {
    property =
       ↪ trimToEncoding(UnicodeProperties.getProperty("General_Category="
       ↪ + propertySpec, true));
} else if (UnicodeProperties.isSupportedScript(propertySpec, true)) {
    property = trimToEncoding(UnicodeProperties.getProperty("Script="
       ↪ + propertySpec, true));
```

```
} else if (UnicodeProperties.isSupportedProperty(propertySpec, true)) {
    property =
        ↪ trimToEncoding(UnicodeProperties.getProperty(propertySpec,
        ↪ true));
} else {
    bailOut("unsupported Unicode property " + propertySpec);
    // So that the property variable is always written to.
    property = CodePointSet.getEmpty();
}
if (negative) {
    property = property.createInverse(Encodings.UTF_32);
}

return property;
```

**Listing 7:** parse Unicode property

#### 4.3.4 Escape

For every escape sequence the `parseEscape()` is called. The elements in the following list can be parsed and turned into a token.

- Anchors (`\A`, `\Z`, `\z`, `\G`)
- Named Backrefence `\k`
- Word Boundaries
- Nonword Boundaries
- Unicode Properties
- Shorthands
- Escape Sequences like `\t`, `\n`, `\f`, `\r`
- Octal Escapes

How the different implementations of Java Regex compared to JavaScript's Regex function will be explained in the following few paragraphs.

#### 4.3.5 Shorthand Character Classes

Shorthands are predefined character classes. Those differ for each system the regular expression is executed on. For example in Java you have to watch out if for `\d` the `UnicodeCharacterClass`-flag is set. If it is set, the Unicode property for `"Nd"` is requested. Otherwise just the constant `Constants.DIGITS` is returned which in the end are just the numbers between `0` and `9`.

For the additional shorthands `\v`, `\V`, `\h` and `\H` the `JavaLexer` returns predefined constants as stated in the api documentation of Java Regex.

#### 4.3.6 Anchor

To provide the support of specific anchors a new token, `Anchor`, has been implemented which can be created via the `Token.createAnchor(value)` as stated in Listing 8. In combination with a new token also a new class `Anchor` has been added inside the `Token`-class. A anchor stores the value it can have as it is listed in Listing 9.

```
public static Token createAnchor(char ancValue) {
    return new Anchor(ancValue);
}
```

**Listing 8:** Representation of createAnchor()-method

```
public static class Anchor extends Token {
    private final char ancValue;

    protected Anchor(char ancValue) {
        super(Kind.anchor);
        this.ancValue = ancValue;
    }

    public char getAncValue() {
        return ancValue;
    }
}
```

**Listing 9:** Anchor Class

Inside `JavaRegexParser` the token is passed and according to its value the abstract syntax tree is filled. In this version of the parser `\G` is not yet supported and terminates the program as it bails out.

### 4.3.7 Word Boundaries

Even though JavaScript and Java both support word boundaries the implementation is a mix of Ruby's boundary implementation and the one of JavaScript's parser. In the lexer the sequence of `\b` or `\B` is detected and a token of the kind `wordBoundary` or `nonWordBoundary` is returned to the parser. In the parser, the boundaries are built and inserted into the ast-builder as listed in Listing 10. When entering the case it is decided if the *Unicode*-flag is set or not. According to that flag the boundary is then built and added to the AST-builder.

```
case wordBoundary:
    if (lexer.getLocalFlags().isUnicode()) {
        buildWordBoundaryAssertion(Constants.WORD_CHARS_UNICODE_IGNORE_
            ↪ CASE, Constants.NON_WORD_CHARS_UNICODE_IGNORE_CASE);
    } else {
        buildWordBoundaryAssertion(Constants.WORD_CHARS,
            ↪ Constants.NON_WORD_CHARS);
    }
    break;
case nonWordBoundary:
    if (lexer.getLocalFlags().isUnicode()) {
        buildWordNonBoundaryAssertion(Constants.WORD_CHARS_UNICODE_
            ↪ IGNORE_CASE, Constants.NON_WORD_CHARS_UNICODE_IGNORE_
            ↪ CASE);
    } else {
        buildWordNonBoundaryAssertion(Constants.WORD_CHARS,
            ↪ Constants.NON_WORD_CHARS);
    }
    break;
```

**Listing 10:** cases word boundary and non-word boundary

The functions `buildWordBoundaryAssertion()` and `buildWordNonBoundaryAssertion()` are the same as in Ruby's implementation. The only difference are the parameters passed to the function as you can see in Listing 10. Those constants are defined in the `Constants` class.

### 4.3.8 Quantifier

Java and JavaScript both support the same greedy and lazy quantifiers. Therefore the function `parseQuantifier()` was able to be extracted into the parent class `BaseLexer`. As it stayed the same as in JavaScript no explanation is added here. In addition, also the part inside the parser remained equal in both parsers.

Possessive quantifiers are not yet supported by the DFA generator of TRegex. Perhaps in future their support will be added to the system.

### 4.3.9 Unicode Characters and Properties

As already described in detail in paragraph POSIX class 4.3.3 Java supports the use of many Unicode categories. To a lot of those POSIX classes an abbreviation exists, e.g.:

`\p{Letter}` → `\p{L}`
`\p{IsLetter}` → `\p{IsL}`

Also `\P{...}` for the negated classes does work with those abbreviations as well.

### 4.3.10 Atomic Groups

Atomic Groups are not yet supported by the DFA generator of TRegex. Perhaps in future their support will be added to the system.

### 4.3.11 Mode modifier / Inline Flags

This feature is not supported in JavaScript, thus it had to be fully implemented in this parser. To be able to let the parser know that it is an inline flag (also called mode modifier in this context), the lexer forwards a new type of token, the `inlineFlag`. This also adds a function called `addInlineFlag(Token.Kind kind, boolean remove, String flags, boolean add)` to the `Token`-class. Calling this function creates an object of the inner class `InlineFlagToken` which consists of three fields: `boolean remove`, `String flags` and `boolean add`. It also contains a constructor and the getters of the fields.

In the lexer a mode modifier is parsed by calling the `parseGroupBegin()` which calls the function `inlineFlag()`. In Listing 11 we see the representation of the code of the function. It checks if the flag is going to be removed or added to the current active flags. Furthermore, it detects if the modified flags are valid for a certain area or an open area till either another mode modifier changes the flags or the regular expression has reached its end. When creating the token the information if the flag is open ended or for a closed area is also added.

```
private Token inlineFlag(int ch) {
    boolean negative = false;
    if (ch == '-') {
        negative = true;
        ch = curChar();
    }

    JavaFlags newFlags = getLocalFlags();

    while (ch != ')' && ch != ':') {
        if (JavaFlags.isValidFlagChar(ch)) {
            if (negative) {
                if (JavaFlags.isTypeFlag(ch)) {
```

```
                    throw syntaxError(JavaErrorMessages.UNDEFINED_
                       ↪ GROUP_OPTION);
                }
                newFlags = newFlags.delFlag(ch);
            } else {
                newFlags = newFlags.addFlag(ch);
            }
        } else if (Character.isAlphabetic(ch)) {
            throw
               ↪ syntaxError(JavaErrorMessages.UNDEFINED_GROUP_OPTION);
        } else {
            throw syntaxError(JavaErrorMessages.MISSING_DASH_COLON_
               ↪ PAREN);
        }

        if (atEnd()) {
            throw syntaxErrorAtEnd(JavaErrorMessages.MISSING_FLAG_
               ↪ DASH_COLON_PAREN);
        }
        ch = consumeChar();
    }

    return Token.addInlineFlag(negative, newFlags.toString(), ch ==
       ↪ ')');
}
```

**Listing 11:** Representation of `inlineFlag()`-method

In the parser and lexer we work with a flag stack `Deque<JavaFlags> flagsStack`. This variable stores all of our flags and new flags can easily be added or removed. Both classes access the same flag stack in order to call on the same flags. In the parser, as listed in Listing 12, the flag stored inside the token is added to the stack. We store the information if the flag is an open flag inside the variable `boolean openInlineFlag`. In case it is not an open flag we push a group where the flag is valid. When the group ends we check if we had an open flag. If not we pop the topmost flags of the flag stack and reset the variable.

```
case inlineFlag:
    openInlineFlag = ((Token.InlineFlagToken) token).isOpen();
    if (!openInlineFlag)
        astBuilder.pushGroup();
    lexer.pushFlagsStack(new JavaFlags(((Token.InlineFlagToken)
       ↪ token).getFlags()));
    break;
case groupEnd:
    if (!openInlineFlag) {
        lexer.popFlagsStack();
        openInlineFlag = true;
    } else if (astBuilder.getCurGroup().getParent() instanceof
       ↪ RegexASTRootNode) {
        throw
           ↪ syntaxErrorHere(ErrorMessages.UNMATCHED_RIGHT_PARENTHESIS);
    }
    astBuilder.popGroup(token);
```

```
    break;
```

**Listing 12:** Pieces of inline flags inside `parse()`-method

#### 4.3.12 Comment

In case the flag "x" (comment-flag) is set, the regex permits all whitespaces and comments in the pattern. In general we can ignore those contents and loop over it until the comment has ended. The same concept applies to the whitespaces. The problem which occured while programming was what to do if the comment is at the end of the regular expression. Using the loop we would run to the end and then check for the next character to return a token back to the parser as in Listing 13. This would lead to an error as we would have reached the end of the pattern.

```
while (getLocalFlags().isExtended() && (ch == '#' ||
   ↪ WHITESPACE.get(ch))) {
    ...
}
switch (ch) {
    case '\\':
        return parseEscape();
    case '|':
        ...
```

**Listing 13:** Concept of loop with comments

In order to fix this error we added a token called `comment`. Inside the parser we detect this case but do not do anything as we ignore comments. The lexer, as in Listing 14, switches to an `if`-statement and either calls the method for comments or consumes a character for a whitespace to get to the next position in the pattern. Both cases then return a token with the kind `comment`.

```
if (getLocalFlags().isExtended()) {
    if (ch == '#') {
        comment();
        return Token.createComment();
    }
    if (WHITESPACE.get(ch)) {
        consumeChar();
        return Token.createComment();
    }
}
```

**Listing 14:** Dealing with comments and whitespaces

### 4.4 Challenges

#### 4.4.1 Setting up the development environment on Windows

Creating such a parser to fully provide functionality of Java Regex being parsed with Truffle is challenging. Before being able to start programming the environment needed to be set up which turned out to be difficult. The machine used to program is a Windows computer and GraalVM is mainly designed for Linux users. Thus, some adjustments needed to made to get the *GraalVM compiler* and co running.

### 4.4.2 Relevant features not implementable in TRegex

It became clear while implementing that some features, which were originally planned to be supported, cannot be supported as the code structure will not allow it (e.g. possessive quantifiers). In addition, the first concept of building a literal string to forward to TRegex was changed to a system creating tokens. Those tokens as described in this chapter are being passed to an AST-builder. Furthermore, towards the end of the project the parser has been split up into a lexer and parser.

# 5 Testing

In this chapter, we will discuss the testing strategy and how it is implemented. Section 5.1 presents the concept of the testing implementation and finally in Section 5.2 we will talk about remaining problems and future work.

In the previous chapters the differences of Java's regex and JavaScript's regex have been shown. Moreover, a solution to mitigate those differences has been laid out. To verify the correctness of the implementation a testing strategy has been chosen. Throughout the implementation of this work, test-driven development was applied. As a first step, tests for all the relevant features as well the differences as described in Section 3.2 have been created. This helped during the implementation phase, as problems could be identified early. As time went by those tests were adjusted and a few tests were added to fine-tune the efficiency and outcome.

## 5.1 Unit tests in Java code

For testing purposes a separate class has been created: `JavaUtilPatternTests`. This class file contains several testing methods and the three essential functions: two times `test()` and `flagsToString()`. Ruby and Python use the `test()`-function of `RegexTestBase` but Java needed a separate one as we do not pass the information if it is a match and where it matches by hand but as we program in Java we let Java do it on its own. Listing 15 shows this concept.

```
void test(String pattern, int flags, String input) {
    test(pattern, flags, input, 0);
}

void test(String pattern, int flags, String input, int fromIndex) {
    Matcher m = Pattern.compile(pattern, flags).matcher(input);
    boolean isMatch = m.find(fromIndex);
    final int[] groupBoundaries;
    if (isMatch) {
        groupBoundaries = new int[(m.groupCount() + 1) << 1];
        for (int i = 0; i < m.groupCount() + 1; i++) {
            groupBoundaries[i << 1] = m.start(i);
            groupBoundaries[(i << 1) + 1] = m.end(i);
        }
    } else {
        groupBoundaries = EmptyArrays.INT;
    }
    test(pattern, flagsToString(flags), input, fromIndex, isMatch,
       ↪ groupBoundaries);
}
```

**Listing 15:** `test()`-methods

The method `flagsToString()` receives the flags as integer parameters and parses it into a `String` so the `JavaFlags` class can interpret it correctly.

To cover a lot of the features of Java Regex several testing functions have been created. The following aspects are covered by this testing environment (examples are also provided for better understanding):

- Dot - `.`
- Alternations - `aa|aa|aa`

- Backslash Escapes - `\^` for ^, ...
- Character Class:
  - Range - `[a-z]`
  - Negated - `[^a-z]`
  - Nested classes - `[ab[cd]ef]`
  - Literal - "`[`" or "`]`"
  - Intersection - `[abc&&[cd]]` or `[abc&&cd]`
- POSIX classes - `\p{Digit}`
- Shorthands - `\w`, `\d`, ...
- Anchors - `\A`, ...
- Word Boundaries - `\b`, `\B`
- Quantifiers - `?`, `+`, `*`, `abc{1,2}`
- Capturing Groups - `(?:abc){3}`
- Backreferences - `(abc|def)=\1`
- Lookaheads - `t(?=s)`
- Lookbehinds - `(?<=s)t`, `(?<!s)t`
- Mode Modifiers - `(?-i)a`, `(?:st)`

A test method can look like Listing 16 but with more content if needed.

```
@Test
public void modeModifier() {
    test("(?-i)a", 0, "a");
}
```

**Listing 16:** Example of testing method

I have added 34 unittests to cover all features listed above. This resulted in 281 source code lines (LOC) which include all the test methods and helper methods in the same class to execute the regular expressions. In total 110 regular expressions have been created. Features TRegex can understand without converting its expressions have been kept few while on the other hand those in need of conversions have been in number more to test the system more extensively.

## 5.2 Passing and failing tests

### 5.2.1 Failing tests

As already mentioned in the previous section a few features are currently not supported. As a consequence, the following, similar and more regular expressions based on the same features are not functioning properly:

- Atomic Group
  `"a(?>bc|b)c"` with input string `"abcc"`
- Anchor
  `"\\G\\w"` with input string `"abc def"`

- Possessive Quantifier
  `"abc?+c"` with input string `"abccdd"`
  `".*+"` with any input string (does not match anything)
  `".++"` with any input string (does not match anything)
  `"a{2,4}+a"` with input string `"aaaaaa"`

### 5.2.2 Passing tests

All the other tests pass. Therefore there are no bugs to the best of my knowledge except those features marked as not available above. In total all 34 tests pass as those failing scenarios are expected to fail and therefore succeed. If those tests would not be marked to fail, 4 out of the 34 tests fail.

# 6 Benchmarking

In this chapter several benchmarks will be run and analyzed. First, in Section 6.1 all the benchmark results are listed and illustrated with its accompanying test cases. In the next part, Section 6.2, those results will be analyzed to find out if there is something where the parser can be improved at, as well as the strengths and lacks of the transpiler from Java to JavaScript are discussed.

## 6.1 Benchmark Results

All the benchmarks are defined in `JavaRegexBenchmark.java` which is stored in the testing environment. Several cases have been thought of and were implemented. Each of them were run after each other and the scores taken of the command line where the benchmark was executed in. The different benchmark functions vary from simply checking for matches (just if it matched at all) in Java and TRegex to checking each of the capture groups found with their indices in Java and TRegex. In following the details of the machine on which the benchmarks were run:

| | |
|---|---|
| **Java Version:** | 11.0.13 |
| **Java Info:** | OpenJDK 64-Bit Server VM GraalVM 22.1.0-dev |
| **JVMCI Version:** | 22.0-b02 |
| **GraalVM Version:** | 22.1.0 (CE) |
| **Graal.js Version:** | 22.1.0 |
| **Machine architecture:** | amd64 |
| **Machine OS:** | Windows 11 |
| **Machine Specs:** | CPU: i5-10400F, RAM: 16 GB, Samsung SSD 970 Evo, GPU: NVIDIA GeForce GTX 1650 SUPER |

After running a few benchmarks - see a sample result in Table 6.1 - it became clear that for Java the benchmark is faster checking for the different capture groups than for just checking for a match for the same regular expression. TRegex on the other hand is way slower when checking for capture groups. This may be caused as we do need to invoke all the parameters to analyze in TRegex and in Java there are built-in functions for the same parameters. As this does distort the scores we will omit capture groups to keep the comparison just.

**Table 6.1:** Benchmark Result Example

| Benchmark | Mode | Cnt | Score | ± Error | Units |
|---|---|---|---|---|---|
| JavaRegexBenchmark.javaCaptureGroups | thrpt | 5 | 21992,840 | 791,036 | ops/ms |
| JavaRegexBenchmark.javaPattern | thrpt | 5 | 19670,025 | 689,706 | ops/ms |
| JavaRegexBenchmark.tregex | thrpt | 5 | 26664,497 | 740,425 | ops/ms |
| JavaRegexBenchmark.tregexCaptureGroups | thrpt | 5 | 13358,874 | 411,274 | ops/ms |

In the Table 6.2 all results of performed benchmarks are presented. The first column shows the regular expression used on the input string presented in the second column. The next columns are showing the score and the error which is the number of deviation between all the tested values as the score is just a mean of the single scores. Those 4 columns are the scores and errors achieved by using Java's own regex engine and the scores by using TRegex. The next column presents the difference between Java Regex and TRegex. Lastly, the last column gives the factor of a speedup when using TRegex instead of Java Regex. The value of the score has the unit `ops/ms` which are the number of operations performed each millisecond.

**Table 6.2:** Benchmark Results

| | | | **Java [ops/ms]** | | **Tregex [ops/ms]** | | | |
|---|---|---|---|---|---|---|---|---|
| # | **Regex** | **Input** | Score | Error | Score | Error | Difference | Speedup |
| 1 | [Hh]ello [Ww]orld! | hello World! | 19670,025 | 689,706 | 26664,497 | 740,425 | 6994,472 | 1,36 |
| 2 | [Hh]ello [Ww]orld! | bigger input as above | 6427,079 | 199,867 | 12476,388 | 296,000 | 6049,309 | 1,94 |
| 3 | .*xy.*ab | ababxyxyxyxyxyxyababxyxyxyxy | 3791,526 | 350,027 | 14914,147 | 497,162 | 11122,621 | 3,93 |
| 4 | .*xy.*ab | bigger input as above | 3033,370 | 64,737 | 5948,494 | 146,082 | 2915,124 | 1,96 |
| 5 | (?<x>abc\|def)=\k<x> | def=def | 18563,210 | 241,526 | 17903,224 | 255,848 | -659,986 | 0,96 |
| 6 | (?<x>abc\|def)=\k<x> | bigger input as above | 2756,619 | 26,179 | 8755,913 | 267,742 | 5999,294 | 3,18 |
| 7 | (?<x>abc){3} | abcabcabc | 16950,162 | 2141,099 | 30320,177 | 629,294 | 13370,015 | 1,79 |
| 8 | (?<x>abc){3} | bigger input as above | 17225,782 | 365,391 | 30388,337 | 530,279 | 13162,555 | 1,76 |
| 9 | (?=(\d+))\w+\1 | 456x456 | 23762,406 | 214,264 | 20840,393 | 500,063 | -2922,013 | 0,88 |
| 10 | (?=(\d+))\w+\1 | bigger input as above | 5954,598 | 202,802 | 4818,060 | 148,763 | -1136,538 | 0,81 |
| 11 | \A(?=\w{6,10}\z)(?=[^a-z]*[a-z])(?=(?:[^A-Z]*[A-Z]){3})\D*\d.*\z | abcd14fDDD | 4162,140 | 51,298 | 24975,361 | 417,528 | 20813,221 | 6,00 |
| 12 | (?<=s\w{1,7})t | ddaaaaadasdtasdf asdsdft | 2973,105 | 106,789 | 5136,151 | 73,632 | 2163,046 | 1,73 |
| 13 | ^([a-z0-9_\\.\\-]+)@([\\da-z\\.\\-]+)\\.([a-z\\.]{2,5})$ | daniel.jaburek@gmail.com | 2839,334 | 32,838 | 14917,468 | 253,624 | 12078,134 | 5,25 |
| 14 | [A-Fa-f0-9]{64} | 9f86d081884c7d659a2feaa0c55... | 1300,327 | 27,009 | 2362,614 | 85,700 | 1062,287 | 1,82 |

In order to calculate an average speedup we want to normalize the scores of Java Regex and TRegex. Thus, we set Java Regex to a value of 1 to see in what relation TRegex stands compared to Java Regex. Taking all speedups in the last column helps us normalize the speedup. This results in an average speedup of **238%** or factor **2,38** of TRegex in comparison to Java Regex. This normalization is illustrated in Figure 6.1. This figure displays both speedup factors as bars in a bar diagram.

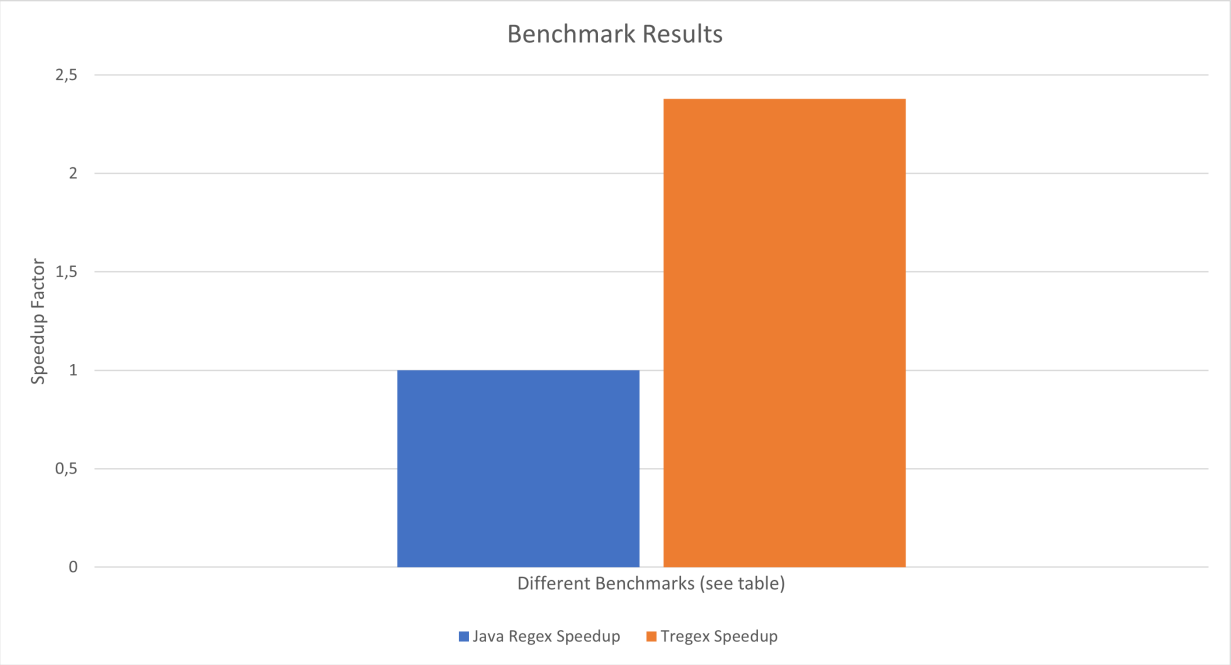



**Figure 6.1:** Average Speedup of TRegex compared to Java Regex

## 6.2 Analyzing the Benchmark Results

On the one hand, as in Table 6.2 and in Figure 6.1 presented TRegex has achieved a higher score than Java Regex doing the same tests. On the other hand, Java Regex has taken the lead in "Backreferences" and "Named Backreferences".

- `(?<x>abc|def)=\k<x>` - Named Backreference
- `(?=(\d+))\w+\1` - Backreference

It appears that Backreferences are not one of the strengths of TRegex as TRegex is a DFA and if it cannot translate or parse a regex it falls back into the backup engine which is a backtracker. This slows the performance down as the test results confirm.

In all other tests as listed in the table TRegex was able to show its higher efficiency with the highest speedup of factor 6 in a test. The average speedup of factor 2,38 makes TRegex more than twice as fast as Java Regex. Regexes including `.`, `*` and character classes are those which were the fastest when running the benchmarks.

The benchmarks run and also future benchmarks can help improve the regex parser of Java but also the regex engine of Truffle as well. The results highlight part of the parser where no performance increase has been accomplished which does not mean that the parser is lacking the implementation thereof. It needs further investigation if the engine can interpret those phrases. Another aspect to consider is how

to built a benchmark. It seems in order to run the TRegex part, reflection is required which can slow the program down as JVM cannot optimize this bit on compilation time.

# 7 Conclusion

The aim of this thesis was to provide access to TRegex from Java applications by creating a bridge between TRegex and the interface `java.util.regex.Pattern`. Furthermore, it should prove the efficiency of the Truffle system compared to Java in regards of regular expressions.

This project started with researching the basics about Graal, Truffle, regular expressions as well as papers and codes of other transpilers in Graal. This enabled creating a list of features Java Regex supports which TRegex does not. Additionally, internally the differences between Java Regex and the regular expressions of Ruby and Python have also been reviewed in order to know if a template of certain features coexists and can be used or if a new algorithm had to be created.

Taking this list of differences helped to develop algorithms for translating and adding of those "new" and different features. It has been taken into consideration if the transpilation is possible, how it can be done in a brief overview and if some work in the same direction has already been done.

While programming, the first concept of parsing a string into another string, which TRegex can interpret, has changed into a concept of using Tokens as described previously in this thesis. It makes the process simpler and faster as the string does not have to go through additional parsing by TRegex to build its abstract syntax tree. The tree is already built while parsing with an AST builder using those tokens. In easy words every literal of the regular expression is taken, looked at, interpreted and turned into a token which is then forwarded to the corresponding AST builder function.

Implementing the source code part and all the classes, algorithms and so on has happened mainly in one class file which resulted into a few thousand lines of code and made it harder to keep an overview of the structure. This lead to separating parts into a *Lexer* and a *Parser*. Briefly said, the lexer is responsible for parsing the regular expression string and creating tokens. Those tokens are passed onto the parser which will call the AST builder functions. After detecting multiple similarities of Java's lexer and the lexer of ECMAScript, which already existed before, a combined base class was implemented to reduce code duplication.

During the investigation of Java Regex and TRegex together with programming it turned out that not all features, which are supported in Java, can be supported in TRegex. Those include possessive quantifiers, atomic groups and the escape term "`G`". This is caused by TRegex not supporting those aspects at the current time.

Whilst and before programming the testing suite has been established and set up to ensure all features are supported in the right way. This helped fasten the programming stage and detect failures along with areas which this transpiler cannot cover currently.

In order to answer the question stated in the beginning "*Is TRegex faster than Java Regex?*" benchmarks have been added. It was necessary to measure the parser with multiple and different regular expressions to try cover the wide area of Java Regex. Measuring how many operations per seconds can be achieved helps comparing it to one another. Normalizing the values and calculating an average score answers the question. This led to the conclusion that TRegex is indeed faster than Java Regex by a factor of 2,38. In some aspects even a speedup of a factor 6 has been reached. But TRegex is not in all areas of regular expressions faster than Java Regex. For example TRegex is slower for regular expressions containing "Backreferences" or "Named Backreferences".

This project has been added to *GraalVM* on GitHub. As not all features are currently supported it keeps future work open. With time progressing TRegex might be integrating atomic groups or possessive quantifiers in future. If so, those aspects can be easily added into the current transpiler. Another open work is to enable easier access to TRegex from Java as at the moment this is only possible through some

lines of code which are still easy to add into a project with regular expressions. But it keeps it open to enhance this access if possible. In addition, TRegex is not yet faster than Java in all aspects as stated in the paragraph above. This adds an area for future work to improve the parser to achieve a speedup in all areas.

# List of Figures

# List of Listings

# References

[1] Dr. Wirth, C. (2018). Dynamic compilation and run-time optimization: Implementing a high-performing interpreter with truffle.

[2] Ecmascript language specification - ecma-262 edition 5.1. (08.12.2021). https://262.ecma-international.org/5.1/#sec-15.10

[3] GitHub. (15.08.2022). Oracle/graal: Graalvm: Run programs faster anywhere. https://github.com/oracle/graal

[4] Goyvaerts, J. (2022). Regular-expressions.info - regex tutorial, examples and reference - regexp patterns. Retrieved July 23, 2022, from https://www.regular-expressions.info/

[5] Haider, J. (2018). An ecmascript 2015-compliant automata-based regular expression engine for graal.js: Master thesis. Retrieved July 23, 2022, from https://epub.jku.at/obvulihs/download/pdf/3053075?originalFilename=true

[6] Matcher (java platform se 7 ). (2020). Retrieved July 23, 2022, from https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html

[7] Openjdk. (20.09.2022). https://openjdk.org/

[8] Pattern (java platform se 7 ). (2020). Retrieved July 23, 2022, from https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

[9] Regular expressions - javascript | mdn. (13.08.2022). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

[10] Regular expressions in java. (06.09.2022). http://www.amygdalum.net/en/efficient-regular-expressions-java.html

[11] Truffle language implementation framework. (15.08.2022). https://www.graalvm.org/22.0/graalvm-as-a-platform/language-implementation-framework/