KING'S College LONDON

University of London

Regular Expressions, Lexing and Derivatives

Final Project Report

Author: Max Hein

Student ID: 1525703

BSc Computer science

Supervisor: Dr Christian Urban

April 16, 2018

Abstract

In 1964, Janusz Brzozowski introduced the notion of derivatives [3]. Derivatives provide a concise way to represent regular expressions. Automata is the default implementation of regular expressions most programmers go to, however like most implementations, has its limitations.
The aims of this project is to implement these derivative regular expressions in a Lexer that follows the algorithm described by Martin Sulzmann and Kenny Zhuo Ming Lu in their paper POSIX Regular Expression Parsing with Derivatives [1]. Following this, implement optimisations to improve the runtime of this proposed lexer.
The goal of the project is to obtain a good understanding of how derivative regular expressions work both within and outside the context of a lexer. Furthermore provide an implementation of the proposed lexing algorithm with optimisations that effectively reduces the time and increases the size of strings a regular expression can match.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Max Hein April 2018

Acknowledgments

I would like to thank my supervisor Dr Christian Urban for all the support and guidance given to me over the duration of this project.

Contents

Chapter 1

Introduction

A string is a list of characters, such that {‘a’, ‘b’, ‘c’} is the string “abc”. To see if a string exists in a sequence of text, algorithms such as the Knuth-Morris-Pratt algorithm were devised [4]. For many processes, it is more useful to match string patterns rather than exact strings, this for example could be searching for all numbers in a sequence of text. To do this, regular expressions are often used. Regular expressions give the notation for patterns to be defined.

Most modern implementations of regular expressions involve the use of Automata. Automata is an abstract machine consisting of a finite number of states and transitions from state to state, such that transitions can be deterministic or non deterministic. Every regular expression has an equivalent deterministic finite automata. The process of converting a regular expression to a deterministic finite automata is done as follows: A regular expression is translated via the Thompson Construction into an epsilon non-deterministic finite automata (*ε*NFA) [6]. This *ε*NFA can then be translated into a non-deterministic finite automata (NFA) by removing all *epsilon*-transitions. Following this, the NFA is converted via subset construction to a deterministic finite automata (DFA) [6]. This is depicted in Figure 1.

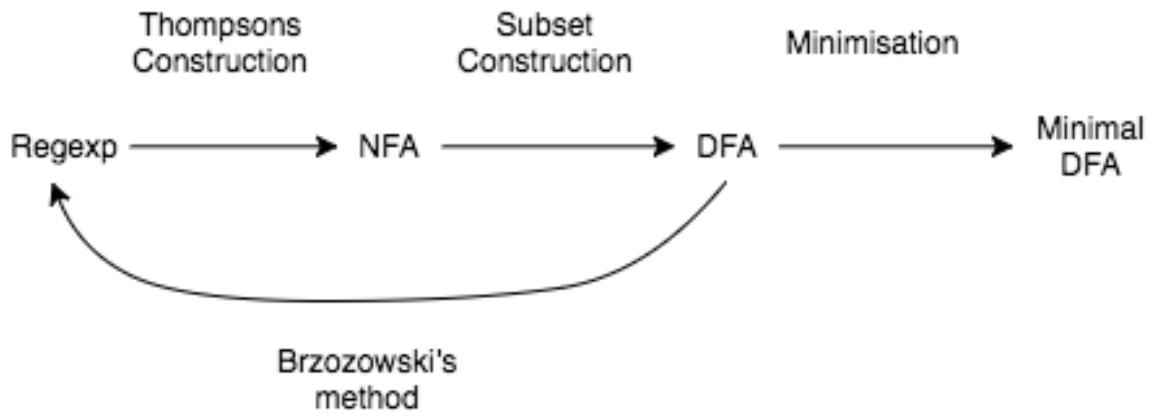

Figure 1: Shows the construction of a DFA from a regular expression

In 1964, Janusz Brzozowski introduced the notion of derivatives. Derivatives offered an alternative way to represent regular expressions. Brzozowski presented a method for identifying string patterns with derivative regular expressions [3]. This was taken further by Martin Sulzmann and Kenny Zhuo Ming Lu in their paper POSIX Regular Expression Parsing with Derivatives [1] where they use Brzozowski matcher in the context of lexical analysis.

1.1 Project aims

The aim of this project is to look into how regular expressions work and how they can be implemented using an alternative method called derivatives. Along with this, implement the proposed lexing algorithm using derivatives, based on the research of Martin Sulzmann and Kenny Zhuo Ming Lu [1]. After implementing these regular expressions in the context of a lexer, the aims of the project become open ended in the sense that different optimisations are to be applied to the lexer, however no specific optimisations are stated.

1.2 Report structure

The layout of this project does not follow the default structure. This is due to this being a research driven project rather than a software development driven project.
The report begins with giving background information on the topics relevant to the problem of lexing. Along with this, current implementations of regular expressions will be further discussed, and an overview of the benefits of derivative based implementations of regular expressions will be made.
The next chapter will cover the design of the proposed lexer. A detailed description of the functions used in the lexing algorithm will be given, along with the functional definitions. Where possible, inductive proofs or referenced papers will be given to prove statements and to put the functions in context. Along with this, different optimisations to the lexer will be described using detailed function definitions. Where appropriate example dry runs of the functions will be made.
The next chapter will cover experimentation, presenting data from lexing different sized inputs with different regular expressions. Any unusual results will be investigated and discussed here. Following this, the data from the experiments and assumptions made in the design are compared and evaluated. The strengths and weaknesses of the algorithm will be identified and discussed.

Finally, a summary is given on what was learned and what was successful about the project, followed by an outline of future work that could be conducted to build on the work in this project.

No requirements or specification section was given in this report due to the open ended nature of the requirements. No implementation section was given in this report due the functional definitions being almost identical to the implemented functions.

Chapter 2

Background

2.1 Regular expressions

Regular expressions are one of the core concepts for this project. They were introduced by Stephen Kleene in the 1950s and are a method for describing regular languages [4]. A language in this context is a set of strings. Regular expressions are used in many text processing tasks such as syntax highlighting, lexical analysis, pattern finding in hostile network traffic, web-crawlers, dictionaries, DNA-data, ad filters etc.

The following are the *basic* regular expressions required to be able to describe regular languages [5].

$$r = 0 \mid 1 \mid c \mid r + r \mid r \cdot r \mid r^\star$$

Each regular expression has a meaning, which can be defined by describing the set of strings they describe. This is known as the *language* and is defined using a function L [5,19].

$L(0) = \{\}$

$L(1) = \{[]\}$

$L(c) = \{"c"\}$

$L(r_1 \cdot r_2) = L(r_1)@L(r_2)$

$L(r_1 + r_2) = L(r_1) \cup L(r_2)$

$L(r^*) = U_{0<n} L(r)^n$

Here 0 represents the empty language. This is a language that consists of nothing, not even the empty string. 1 represents a language containing only the empty string. c represents a language consisting of a single character, where c could represent any character. The concatenation of two languages A and B can be defined as follows: $A@B = \{s1@s2 \mid s1 \in A \wedge s2 \in B\}$, therefore each string in A is joined with every string in B. (r1 · r2) is a sequence of regular expressions. The language

defined by (r1 · r2) is the concatenation of the language of r1 and the language of r2. The union of two languages r1 and r2 is a new language r3, where the language of r3 consists of all the strings in r1 and all the strings in r2. (r1 + r2) represents an alternative or choice between the left regular expression r1 and the right regular expression r2. The language it represents therefore is the union of the two languages. r* describes a language consisting of the empty string and the concatenation of one or more of the languages described by r, more formally [5]:

$L(r)^0 = \{[]\}$

$L(r)^{n+1} = L(r) \mathbin{@} L(r)^n$ where concatenation is redefined to $\{s_1 @ s_2 \mid s_1 \in L(r) \wedge s_2 \in L(r)^n\}$

Regular expressions can now be formulated using these definitions to obtain a description of a regular language. For example ('a' + 'b') is the regular expression who's language consists of 'a' and 'b'.

Most programming languages offer libraries that can be used to check if a given string is an element of a language as described by a regular expression. Along with the above basic regular expressions, the extended regular expression set is also usually given [18].

r = r+ | r? | r{n} | r{n,m} | […]

And again can be defined using the function L:

$L(r+) = L(r) \mathbin{@} L(r^*)$

$L(r?) = \{[]\} \cup L(r)$

$L(r\{n\}) = L(r) \mathbin{@} L(r) \mathbin{@} \ldots \mathbin{@} L(r)$ L(r) is repeated n times

$L(r\{n,m\}) = L(r\{n\}) \cup L(r\{n+1\}) \cup \ldots \cup L(r\{m-1\}) \cup L(r\{m\})$

$L([\ldots]) = \{\ldots\}$ where … is a set of characters

Different programming languages will have different symbols to represent these operations, but will most likely have these operations in some form. The extended regular expressions provide a more concise way to represent longer regular expression statements however, give no additional descriptive power to regular expressions. This is with the exception of backtracking regular

expressions which is not covered in this report. As regular expressions describe regular languages, there is no regular expression for describing the language of n a's followed by n b's, where n is the number of a's, followed by the same number of b's also described as (a^n . b^n). This is because (a^n . b^n) is not a regular language. It can be proved that this is the case by using the Pumping Lemma [17]. Alternatively they do provide a convenient way to describe URL's. A regular expression for a URL could be, for example:

$$url = w{\cdot}w{\cdot}w{\cdot}.{\cdot}[a\text{-}z]+{\cdot}.{\cdot}c{\cdot}o{\cdot}m{\cdot}:{\cdot}[0\text{-}9]\{4\}{\cdot}(/{\cdot}.{\cdot}[a\text{-}z])+$$

A string is matched by a regular expression if the string is a part of the language the regular expression describes. This can be defined as $s \in L(r)$, where s is the string and L(r) is the language of the regular expression r. The following URL's are a part of the language url:

$www.example.com \in L(url)$

$www.exampleone.com:8080 \in L(url)$

$www.exampletwo.com:8080/example/resource.html \in L(url)$

But the following are not elements of the language of url:

$example.com \notin L(url)$

$http://www.example.com \notin L(url)$

$www.example.com/hello.png \notin L(url)$

Research into regular expressions has been going on for around 60 years, however an interesting characteristics of the subject still causes problems for programmers who implement regular expressions. The characteristic is known as an 'evil regular expression'. Evil regular expressions cause regular expression matchers to run in exponential time when asked to match strings of increasing size and can cause catastrophic backtracking. As a result, given a matcher m that is not optimised to handle evil regular expressions and m tries to match a string to a evil regular expression, m will run slowly and not be able to complete it's matching operation. Evil regular expressions can therefore cause some serious problems, for example, in web-application. Attackers are able to look for instances where the matching engine behaves badly and then mount a Regular

Expression Denial of Service Attacks (ReDoS) against the application [4]. As a result, paralysing the availability of the application and thus stoping other users using the service.

2.2 Matching choice

An interesting observation of regular expression matching, is that in some instances, regular expressions are able to match strings in more than one way. For example, given the regular expression r = (a | b | ab)* and string s = ab, r is able to match s in two ways. The first by matching a then b (POSIX), and the second by matching ab (Greedy). As a result, we have a choice on the method in which we match the regular expression. The two main approaches here are Greedy matching and POSIX matching [19]. For this project a POSIX matching strategy will be followed, where the rules for such a matching strategy can be defined as [19]:

The Longest Match Rule / Maximal Munch Rule: The longest initial substring matched by any regular expression is taken as the next token.

Priority Rule: For a particular longest initial substring, the first regular expression that can match determines the token.

The above definitions where taken from the paper POSIX Lexing with Derivatives of Regular Expressions [2]. The greedy matching strategy is affected by the order of alternative statements. Where as the POSIX matching strategy gives preference to the largest matched string and thus is more complicated as a dependency is created for information about the length of matched strings [19].

2.3 Current implementation

Regular expressions as described in the background section can be implemented using automata, as every regular expression has an equivalent deterministic finite automata (DFA). A DFA can be obtained through Thompsons Construction followed by Subset Construction [6]. The resulting DFA can then be traversed resulting in an accepting or rejecting state being reached. This method of implementing regular expressions is the basis of most current regular expression matchers. Using automata however can be very slow due to the fact that the conversion of an NFA into a DFA can produce 2^n states and therefore matching with such a DFA could take considerable time. For the same reason it can be impractical to produce a DFA for large NFA's. The case is worsened when

considering that DFA minimisation is an NP-complete problem and thus a deterministic polynomial-time algorithm has not yet been discovered. Even if DFA minimisation was not an NP-complete problem, the DFA produced from the minimisation could still be exponential in size. Alternatively NFA backtracking could used, however again this has limitations. As a result, developers of regular expression matchers are always looking for different techniques to match regular expressions that are both quick and reliable when considering evil regular expressions.

2.4 Derivative implementation

Another proposed way to implement regular expressions is a notion called derivatives. The use of derivatives in the field of regular expressions was introduced by Janusz Brzozowski in 1964 [3] and where, to quote the paper Regular-expression derivatives re-examined [7], 'lost in the sands of time'. Brzozowski proposed a regular expression matcher that used this notion of derivatives instead of using automata. It is a simple idea. The derivative function will take a string and a regular expression. Then iterating the characters of the string, it will output a new regular expression that matches all the strings defined by the original regular expression that start with the given character, but with the first character removed. This is repeated for all characters in the string. If at the end the regular expression returned can match the empty string, the regular expression must match the input string. This is described in detail in the next chapter of this report.

Derivatives are defined using a functional notation and thus are easy to implement in functional programming languages such as Scala, F#, ML, Haskell etc. Its functional definition allows for additions to easily be made and or later made to the implementation. To implement a new regular expression, only the addition of the expression case needs to be added to the already defined functions, and would not require the alteration of any prior code - for example, adding the extended regular expression set, given only the basic regular expressions are implemented. Their functional definition also make derivatives easier to reason with in theorem provers such as ISABELLE [8]. As a result, it is possible to prove the regular expressions operate as they are intended to. This is important as Kuklewicz observes nearly all POSIX-based regular expression matchers are buggy [22]. For companies that implement these *Matchers* in systems such as *Lexers*, they can be confident of their operational soundness. This is an important selling point when using this software in environments such as space and medicine. This is a great advantage for derivative base

implementations of regular expressions over automata, which is difficult to reason with in theorem provers due to its graphical nature.

2.5 Lexer

A *Lexical Analyser* (also known as a *Lexer* or *Tokeniser*), processes a sequence of characters such as a programming language into a sequence of *tokens*. Tokens are a 2-tuple of strings. This is done by using regular expressions to match/identify sequences of characters in the given input. Following the POSIX rules [19], the largest possible substring is matched to a given regular expression that represents an instance, for example a keyword in a programming language. This process is repeated until all characters in the input have been matched. Consider a statement in a programming language. It is required to be able to identify all the elements that make up the statement.

For example:
Given the following statement:

int num = 3 + 3

The tokens produced could be of the form:

[(keyword, int), (identifier, num), (operator, =), (number,3), (operator, =),(number,3)]

A lexer is generally combined with a p*arser*, where the output of the parser is a distinct abstract syntax tree that can be used in processes such as compilation or interpretation. In this instance most current programming languages implement this process using push down automata however, as with lexing, there is more than one way to implement a Parser of which Parser Combinators is an alternative solution. In some instances a lexer and parser are combined into one program leading to terms such as *scanner-less parsing* or *lexerless parsing*. The use of lexing can be extended to other tasks, an example that seems trivial at first could be the removal of white spaces in a section of text. Lexing can be implemented in many ways, such as the Sulzmann & Lu algorithm [1] or using automata as implemented in most current languages. Most programmers however don't implement lexers from scratch for their projects. There are many libraries available that implement lexers for example GNU for the C language [9]. When programmers are implementing e.g. bespoke compilers, they will usually turn to these libraries for lexer implementations.

Chapter 3

Design

The general concept of a lexer is as follows. A lexer takes a string, a set of regular expression rules and outputs a list of tokens that tell us what elements make up the string, for example: Identifier, Keyword, String, Number etc. In this project a lexer is implemented based on the use of derivatives, as defined by Martin Sulzmann and Kenny Zhuo Ming Lu in their paper POSIX Regular Expression Parsing with Derivatives [1]. Due to the algorithm already being functionally defined in their paper, this section will be used to describe these algorithms in a more abstract way to obtain a better understanding of what each function is doing.

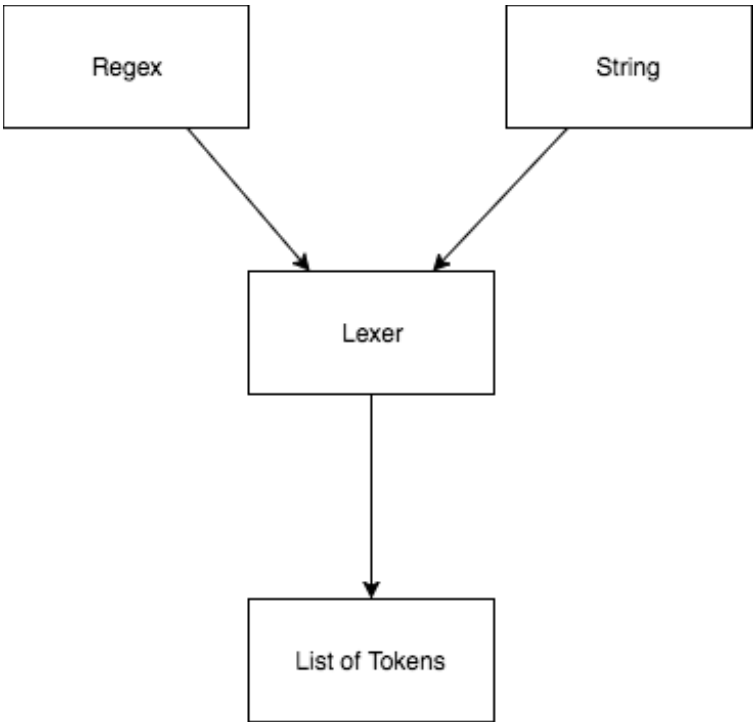

Figure 3 shows the general model of a lexing programme, defining the inputs and the result of the programme.

3.1 Derivative based Lexing algorithm

The following algorithm is taken from the paper POSIX Regular Expression Parsing with Derivatives by Martin Sulzmann and Kenny Zhuo Ming Lu [1]. The functional definitions and naming conventions are taken from the Martin Sulzmann and Kenny Zhuo Ming Lu, POSIX Regular Expression Parsing with Derivatives paper [1] and the POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl) paper [2].

The lexing algorithm is split into two parts, the matching phase and the injection phase. The matching phase is an implementation of Brzozowski Matcher [3], where regular expressions are matched using derivatives. The injection function then produces a *value* for how a string was matched by the derivative regular expression.

3.1.1 Matching phase

A regular expression matcher should give a boolean output based on if the given string is an element of the language described by the regular expression, such that $s \in L(r)$, where s is the string and L(r) is the language the regular expression r describes. The matcher proposed by Brzozowski [3] is again broken down into two parts. The first part of the matcher is the function nullable. The nullable function takes as an argument a regular expression and outputs a boolean value of true or false, based on if the given regular expression is able to match the empty string. A recursive definition of the function is as follows [10]:

$$\text{nullable}(0) = \text{False}$$
$$\text{nullable}(1) = \text{True}$$
$$\text{nullable}(c) = \text{False}$$
$$\text{nullable}(r1 + r2) = \text{nullable } r1 \vee \text{nullable } r2$$
$$\text{nullable}(r1 \cdot r2) = \text{nullable } r1 \wedge \text{nullable } r2$$
$$\text{nullable}(r^\star) = \text{True}$$

The property that Martin Sulzmann and Kenny Zhuo Ming Lu [1] are trying to achieve from the function, is if the regular expression r describes a language containing the empty string return true, else return false, more formally:

nullable(r) if and only if [] $\in L(r)$

'*nullable(r)'* represents the function we can implement and 'if [] $\in$ *L(r)*' represents the specification of the function, 'what we want to achieve'. It can be seen that this property holds for all lines in the function through the use of induction where nullable(0), nullable(1) and nullable('c') are base cases.

nullable(0) iff [] $\in L(0)$: nullable(0) by the above definition is False, thus obtaining False for the left hand side of the property. The language of the empty language is just the empty language therefore the empty string is not an element and the right side of the property is False. As both the left and right parts of the property are False, nullable(0) = False holds.

nullable(1) iff [] $\in L(1)$: nullable(1) by the above definition is True, thus obtaining True for the left hand side of the property. The language of the empty string is just the empty string therefore the empty string is an element and the right side of the property is True. As both the left and right parts of the property are True, nullable(0) = True holds.

nullable(c) iff [] $\in L(c)$: nullable(c) by the above definition is False, thus obtaining False for the left hand side of the property. The language of 'c' is just the character 'c' therefore the empty string is not an element of the language and the right side of the property is False. As both the left and right parts of the property are False, nullable(c) = False holds.

nullable (r1 + r2) iff [] $\in$ L(r1 + r2):

Can assume the following holds due to the base cases holding

nullable(r1) iff [] $\in$ L(r1) ,

nullable(r2) iff [] $\in$ L(r2)

nullable (r1 + r2) by definition is nullable(r1) $\vee$ nullable(r2)

nullable(r1) $\vee$ nullable(r2)

[] $\in$ L(r1) $\vee$ [] $\in$ L(r2)

[] $\in$ L(r1) $\cup$ L(r2)

[] $\in$ L(r1 + r2) (By the definition of L as described in the introduction)

nullable (r1 · r2) iff [] ∈ L(r1 · r2):

Can assume the following holds due to the base cases holding

nullable(r1) iff [] ∈ L(r1) ,

nullable(r2) iff [] ∈ L(r2)

nullable (r1·r2) by definition is nullable(r1) ∧ nullable(r2)

nullable(r1) ∧ nullable(r2)

[] ∈ L(r1) ∧ [] ∈ L(r2)

[] ∈ L(r1) @ L(r2)

[] ∈ L(r1 · r2) (By the definition of L as described in the introduction)

nullable (r ⋆) by definition is True. By definition [] ∈ L(r*). As a result both sides are True making the statement True.

The second part of the matching algorithm calculates a derivative of a regular expression through the use of the function der. If thinking of regular expressions as the set of strings they describe, the derivative function performs a transformation to the set of strings they describe, this function will be defined as derl. The transformation results in a set that contains all the strings that start with the given character, except with the first character removed. For example A = {aone, btwo, cthree, afour} then given c is the character, the set returned would be {one, four}. As der works on regular expressions, not sets of strings, given L is a function that represents the language of a regular expression, L(der a r) = derl c L(r).

Der takes as arguments a regular expression r, a character c and returns a new regular expression. This new regular expression can match a string of the following form, if *r* can match a string of the form *c* :: *s*, the new regular expression will match just s.

Der is defined as follows [10]:

der (c, 0) = 0

der (c, 1) = 0

der (c', c) = if c' = c then 1 else 0

der (c, (r1 + r2)) = der(c r1) + der(c r2)

der (c, (r1 · r2)) = if nullable(r1)

then (der(c r1) · r2) + der(c r2)

else der(c r1) · r2

der (c, ($r^\star$)) = der(c r) · ($r^\star$)

The regular expressions 0 and 1 are unable to match a string of the form c :: s as the languages they represent are the empty language and a language consisting of only the empty string. As a result 0 is returned, thus showing that the given regular expression can't match this input/string. For the third statement where a character regular expression is given, a decision on whether the character matches the regular expression is made. If they match, return 1, this is because given a regular expression matches one character and the derivative of this is required, (a regular expression that matches the string s if given a string of the form c::s) only the empty string will be matched by the derivative regular expression. If the character does not match, return 0. For the alternative case, two regular expressions are given (r1 + r2), all strings of the form c :: s are matched by either r1 or r2. As a result, a recursive call of der is made on each of the regular expressions and the results are appended by the alternative (+) regular expression. For the fourth case, if a sequence of regular expressions ($r_1 \cdot r_2$) matches a string of the form c :: s, the first regular expression in the sequence r_1 must have matched c. But r_1 can also match the empty string, in which case, either the first regular expression r_1 or second regular expression r_2 matched c. Therefore, nullable must be used to check if the first regular expression r_1 matches the empty string. If r_1 matches the empty string it has to be considered that r2 can match c thus (der(c r1) · r2) + der(c r2). If r_1 is not nullable, only r_1 needs to be considered to match c, therefore der(c r1) · r2. For the last case of r*, if r* matches a string of the form c :: s, c must have been matched by r. When it is said r matches c it is meant a single copy of r* matches c. As a result der is called on a single copy of r. r* is appended to the end.

All components for Brzozowski Matcher [3] have now been defined. To implement, the der function must be extended to include a string instead of only a single character, we call this new function ders. Ders takes as arguments a string, a regular expression and outputs a new regular expression. Ders recursively calls itself. In each recursive call, one character from the original string is used in a der function call, with the current partial derivative regular expression. The rest of the string is passed to the recursive call along with the result of the der function just made. Recursion stops when the string is exhausted. For example, if matching "abc" with a regular expression r, the ders function will iterate der, thus building the following statement der(c, der(b, der(a, r))). Ders is recursively defined as follows [5]:

ders (c::s) r = ders s (der c r)

Now using the function *ders,* Brzozowski matching algorithm [3] can be defined for a string s and regular expression r [5]:

matches s r = nullable(ders s r)

matches will return true or false based on the result of nullable. The goal of this matches function is to satisfy the following property:

matches s r if and only if $s \in L(r)$

Figure 3.1 gives a diagrammatic overview of the matching process.

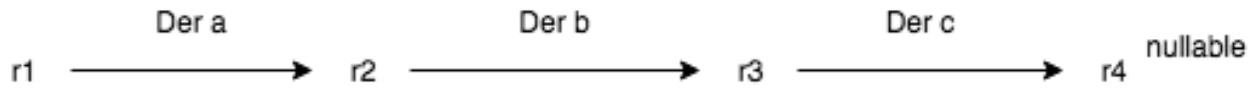

Figure 3.1: Brzozowski matching algorithm [3]

3.1.2 Injection phase

Injection is the next step of the algorithm described by Martin Sulzmann and Kenny Zhuo Ming Lu [10]. So far a matching algorithm has been described which returns true or false depending on if a string can be matched by a regular expression. Now an understanding of how the string was matched is required to then be able to *tokenise* it.

To be able to define how a regular expression matches a string, Martin Sulzmann and Kenny Zhuo Ming Lu introduced the notion of a *Value*. Values are defined as follows [10]:

$$v = \text{Empty} \mid \text{Char } c \mid \text{Left } v \mid \text{Right } v \mid \text{Seq } v1\ v2 \mid \text{Stars } vs$$

Each Value corresponds to a regular expression. The regular expression 0 is not represented by a value as it does not match a string. 1 corresponds to Empty. c corresponds Char(c). (r1· r2) corresponds to Seq v1 v2. (r1+r2) corresponds to both Left v1 and Right v2. r* corresponds to Stars vs where vs is a list of *values*, one for each copy of *r* that was needed to match the string.

Now given a regular expression r = ((c + b) + c) and a string ‘c’ a *value* for how this regular expression r matches the string ‘c’ is Left(Left(Char c)). Here it is important to remember the order of choice on how a regular expression is matched, if it is possible to match a string in more that one way. The choice of how we match the string is based on the POSIX rules [19].

The second phase of the algorithm is organised so that it will calculate a *value* for how the derivative regular expression has matched a string. It is important to reiterate here that the goal of the second part of the algorithm is to obtain a value that defines how a regular expression matches a string, not to re-obtain a regular expression. This is done using two functions, mkeps and inj. mkeps calculates a value for a regular expression that matches the empty string and inj calculates a value for how a regular expression matches a given character. In context, mkeps and inj are shown by the system diagram in figure 3.2, by the green highlights.

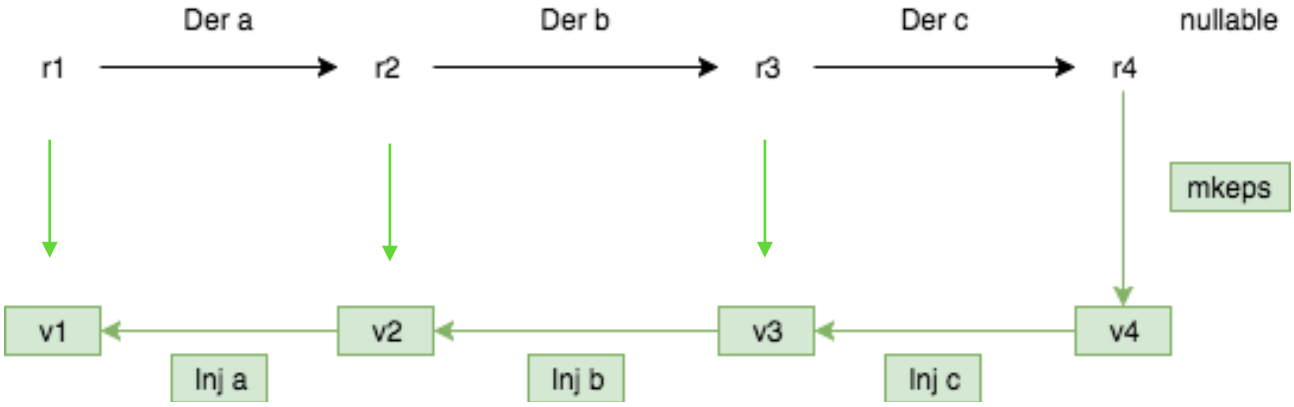

Figure 3.2: Martin Sulzmann and Kenny Zhuo Ming Lu
Lexing algorithm [1], emphasising the Injection phase

From figure 3.2, r4 represents the result of ders, with the input of "abc" and a regular expression that matches this. Therefore the regular expression iterated through the character a, then b and finally c. As a result r4 is a regular expression that once matched abc however, now matches the empty string. The goal for inj and mkeps is to represent how this regular expression matched "abc" using values.

The *mkeps* function calculates a value for how a regular expression has matched the empty string. It takes as a parameter a regular expression, which is obtained from the result of ders, and returns a value. mkeps is defined as follows [10]:

mkeps (1) = Empty
mkeps (r1 + r2) = if nullable r1 then Left (mkeps r1) else Right (mkeps r2)
mkeps (r1 · r2) = Seq (mkeps r1) (mkeps r2)
mkeps (r ⋆) = Stars []

There are no cases for 0 and c as these regular expressions cannot match the empty string and thus no value can be calculated. As a result, the function will not come into contact with such expressions. 1 is a regular expression that matches only the empty string, thus mkeps returns Empty. For a choice between two regular expressions, also know as an alternative statement, the empty string is matched by either the left or right regular expression. The alternative case in mkeps gives

preference to the regular expression on the left-hand side, this choice of preference is given by the POSIX rules [19]. For a sequence of regular expressions to match the empty string, the empty string must be in the language of all the regular expressions in the sequence, therefore mkeps calls itself on each regular expression in the sequence. The star regular expression matches the empty string by default as it is an element of every language it describes. Thus, the corresponding value is returned with no values in its list. mkeps is required, as at the end of the matcher a regular expression that matches the empty string is produced. To be able to inject the characters from the string, a value for how the regular expression matched the empty string must first be calculated. This gives the initial value used in the injection function.

inj is similar to mkeps, however injects a given character based on how the regular expression has matched the string so far (as given by the value). Inj takes 3 arguments. The first argument is a character. It is obtained from the original string used in the matching process. If iterating inj the characters are input in reverse order. E.g. if “abc” is the string, inj would first inject c then b and finally a. Next is a regular expression which is obtained from the corresponding derivative step. If injecting for example the character ‘c’ from the string “abc” the regular expression resulting from der(b,(der(a,“abc”))) is used, or if looking at Figure 3.2, r3. The third parameter is a value, it is obtained from how the regular expression has matched the string so far. Figure 3.1.5 shows how values are passed from inj to inj along with all the major interactions in this step of the algorithm. The initial value is obtained from mkeps. inj returns a new value and is defined as follows [10]:

inj d c Empty = Char d
inj (r1 + r2) c (Left v1) = Left (inj r1 c v1)
inj (r1 + r2) c (Right v2) = Right (inj r2 c v2)
inj (r1 · r2) c (Seq v1 v2) = Seq (inj r1 c v1) v2
inj (r1 · r2) c (Left (Seq v1 v2)) = Seq (inj r1 c v1) v2
inj (r1 · r2) c (Right v2) = Seq (mkeps r1) (inj r2 c v2)
inj (r ⋆) c (Seq v (Stars vs)) = Stars (inj r c v :: vs)

There is no case for 0 or 1 as a character can’t be matched by these regular expressions and thus can’t be injected into these positions. If a character regular expression matched the given character, the associated value of Char is returned. There are two cases for the alternative regular expression (r1 + r2) as either the character was matched by r1 or r2 and thus will be injected as a Left or Right

value. How this information is known is based on the given value. There are three cases for the sequence regular expressions (r1 · r2), as there are three possible values this regular expression can encounter. Again, it is possible for the first regular expression (r1) to match the empty string. This means that both r1 and r2 can match the input. The input is matched by r2 if r1 matches the empty string, otherwise the input is matched by r1. If the given value is a sequence, the character is injected into r1, based on the first value in the sequence. The resulting value is a sequence of the result of the injection and the second value in the original sequence. If a Left value is given containing a sequence of two values, the character is again injected into r1 with the first value in the sequence. The resulting value again is a sequence of the result of the injection and the second value in the original sequence. The third case for the sequence regular expression is if a Right value is passed. In this case, mkeps is used to get a value for how r1 matched the empty string and the character is injected into r2. The result is a sequence of these two results. The star regular expression matches the given character with one copy of its self, r. Inj returns a list of values where the the first value is used to inject the given character into r. The rest of the values are added to the result of the injection to get the sequence of values.

All the above functions where defined as parts used in a lexing algorithm that can now be defined as follows [10]:

```
lex r []    = if nullable(r) then mkeps(r)
              else error
lex r c::s = inj r c lex(der(c, r), s)
```

Similar to ders, lex iterates inj and allows it to perform injection on a string. If the given string is empty, the given regular expression must match the empty string to be able to match the input. This is checked using the nullable function. If it is able to be matched, a value for how the given regular expression matched the empty string is calculated using mkeps. If looking at Figure 3.2 it is the case where there is a transformation from r4 to v4. If the given string is not the empty string, the characters are used to obtain a derivative regular expression. The derivative is calculated in a nested manner until only the empty string can be matched. If looking at figure 3.2 it is the steps from r1 to r4. The resulting derivative regular expressions are then used in the injection function to inject the characters back into the expression.

Figure 3.3 is a diagrammatic representation of the two phases of the algorithm operating on the input string “abc” and a regular expression that matches such a string.

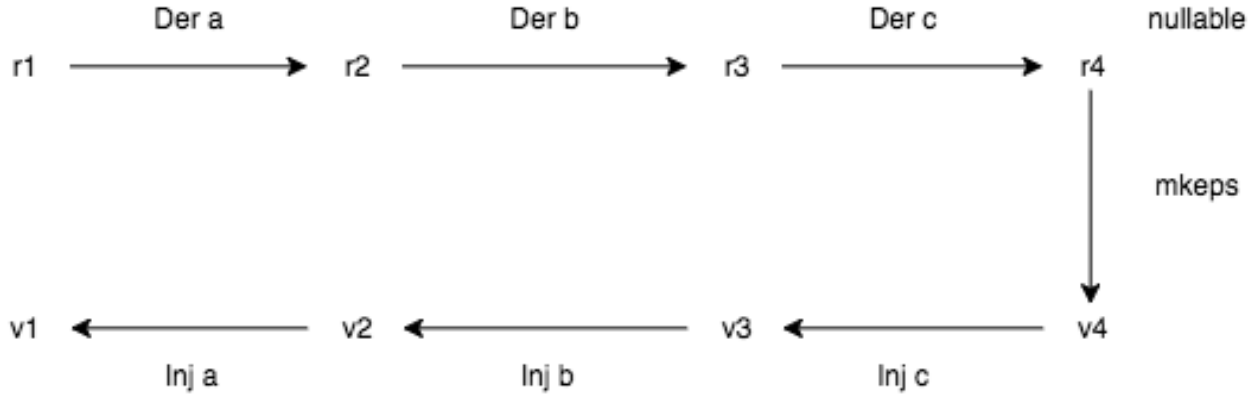

Figure 3.3 Martin Sulzmann and Kenny Zhuo Ming Lu
Lexing algorithm [1] represented diagrammatically

3.1.3 Tokenising

Inj returns a value of how a regular expression matches a given input. To split strings into tokens a simple modification to this algorithm must be made. As described in the introduction, tokens are a 2-tuple. In this implementation it is a 2 tuple of strings. An example input that would need to be tokenised is a URL. Therefore splitting the URL into domain name, port number, resource path, application layer protocol etc. To do this, an additional regular expression and value is required. They are defined as the *record regular expressions* and *record value* [6]. They allow for the concentration on certain parts of a regular expression. RECD is the record regular expression and Rec is the record value. As record is a regular expression nullable, der, mkeps and inj have to be extended. They are defined as follows [11]:

r = RECD(x : r)

v = Rec(x : v)

nullable RECD(_, r1) = nullable(r1)

der RECD(_, r1) = der(c, r1)

mkeps RECD(x, r) = Rec(x, mkeps(r))

inj (RECD(x, r1), _) = Rec(x, inj(r1, c, v))

Given the regular expression a·b + a·c. When tokenising, it is not required to know if the regular expression recognises the given string. The only information required is if it matched and if it did what part of the string is of interest. In this example either ab or ac can be matched by a·b or a·c and only the last character is of interest. The last characters they can match are ‘b’ or ‘c’. To obtain only the last character (or the part of interest) the records regular expression is used. The new regular expression looks as follows:
a . (x : b) + a . (x : c), where x is the identifier/name of the sequence of characters the recorded regular expression represents, for example keyword, number etc.

To put in context given a URL:

www.example.com:1030/example/page.html

Here, the host name, port number and resource path needs to be identified. The regular expression to identify a URL is:

w·w·w·.·[a-z]+·.·c·o·m·:·[0-9]{4}·(/·.·[a-z])+

To identify the host, port and path using records, the additions to the above regular expression is as follows:

(Host : w·w·w·.·[a-z]+·.·c·o·m) · : · (Port : [0-9]{4}) · (Path : (/·.·[a-z])+)

Obtaining the tokens: {(Host : www.example.com), (Port : 1030), (Path : /example/page.html)}

To extract these recorded identifiers a function called env is implemented. Env takes a value and outputs a list of tokens. Env is defined as follows [11]:

env(Empty) = []

env(Char(c)) = []

env(Left(v)) = env(v)

env(Right(v)) = env(v)

env(Seq(v_1, v_2)) = env(v_1) @ env(v_2)

env(Stars[v_1,...,v_n]) = env(v_1)@...@env(v_n)

env(Rec(x : v)) = (x : |v|) :: env(v)

When lexing a regular expression with a record regular expression the value is calculated following the same logic as previously defined. The recorded regular expression is converted into the appropriate value, where it still references the recorded information same as the recorded regular expression did. Env iterates through this value and extracts the records. If the given value is Empty, no record is identified therefore the empty set is returned. If a character value is given, no record is identified, therefore the empty set is returned. If a Left value is given, there may be a record in this value, thus env is called on the given value. This is the same case for if a Right value is given. If a Seq value is given, a record could be anywhere in the sequence of values therefore env is called on every value in the sequence. If a Stars value is given, like the Seq value, a record could be in any value in the list. As a result, env is called on every value in the list. If env encounters a record the underlying string defined by the value is calculated. This is done using the helper function flatten. As there can be records in the value given by the record, e.g. (x:a+(x:b)), env must be called on the value. The results are appended as the result of env is a list. flatten takes as an argument a value and calculates the underlying string. flatten is defined as follows [2]:

flatten Empty = “”

flatten Chr(c) = c

flatten Left(v) = flatten(v)

flatten Right(v) = flatten(v)

flatten Sequ(v1, v2) = flatten(v1) + flatten(v2)

flatten Stars(vs) = vs.map(flatten).mkString

flatten Rec(_, v) = flatten(v)

3.2 Optimisation

It can be observed from figure 3.4 that the regular expression a** is able to match 20 a's in 1.55 seconds when using the lexer proposed by Martin Sulzmann and Kenny Zhuo Ming Lu [1]. To put this in context the same regular expression was run in Python. Python was able to match 20 a's in under 0.1 seconds. Looking at multiple regular expressions it can be concluded that the derivative based lexer is slower than other implementations. As a result, it is not useable outside the context of theory. By making the derivative based lexer more efficient it becomes more useable and applicable to applications.

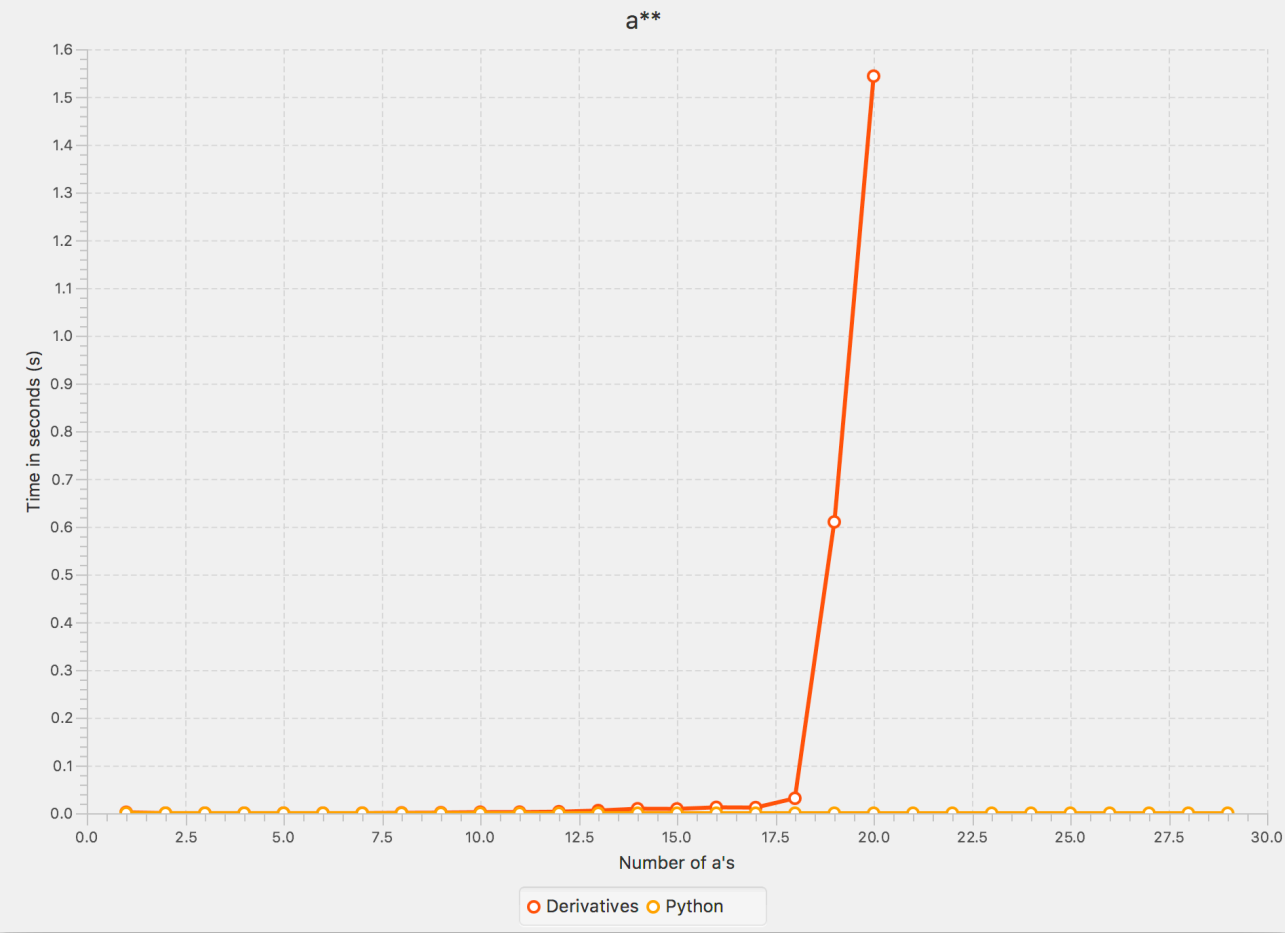

Figure 3.4: Displays the time it takes to match a** in Python and using the derivative based Matcher [3]

3.2.1 Simplification function 1

The size of a regular expression is the number of sub-expressions it is made up of. For example, using our basic regular expressions we construct r = 1 + ‘A’, the size of r is 3. It is easier to see why the size is such when the regular expression is written as defined by its implementation, ALT(ONE,CHAR(‘A’)), as you can see 3 regular expressions are used in the definition.

The optimisation performed is simplification. Simplification of a regular expression is the process of reducing the size, such that it still matches the same language as the unsimplified regular expression, they are ‘equivalent’. More formally given a regular expression R1, a simplification function S and a function L such that L(R1) represents the language of R1; $L(R1) \equiv L(S(R1))$. The simplification function in this section is introduced and defined in the paper POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl) [20]. In this report it is redefined as simp1.

Some basic simplifications for regular expressions are as follows [20]:

$r \cdot 0 \rightarrow 0$

$0 \cdot r \rightarrow 0$

$r \cdot 1 \rightarrow r$

$1 \cdot r \rightarrow r$

$r + 0 \rightarrow r$

$0 + r \rightarrow r$

$r + r \rightarrow r$

Therefore given r = (‘A’ · 1) + 0 and using our simplification function S from above that implements the simplification rules, we get S(r) = ‘A’. To note, L(r’) = L(S(r’)) and ideally Size(r’) > Size(S(r’)).

Simplification is performed at each derivative step, therefore trying to keep the size of the derivative regular expression small, thus increasing efficiency. Efficiency is increased because less expressions have to be stored in memory and also less regular expressions must be processed. Simplification however results in the algorithm building a value for the simplified regular expression, not the original regular expression. This is fine when just matching the string, however

for lexing, it is required to know how the original regular expression matches the string. To overcome this, rectification functions are introduced to re-build the value for the original regular expression. The rectification function takes as argument a value and returns a new value. This new value tells us how the original regular expression matched the input string.

This simplification function is based on the above basic simplifications. It is split into two cases. Instances of alternative statements and instances of sequence statements. It takes as input a regular expression and outputs a simplified regular expression and a rectification function. This simplification function is defined in the paper POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl) [20]. The definition is as follows [20]:

simp1($r_1 + r_2$)
 (r_{1s}, f_{1s}) = simp1(r_1)
 (r_{2s}, f_{2s}) = simp1(r_2)
 if $r_{1s} = 0$ then return (r_{2s}, $\lambda v.$ Right($f_{2s}(v)$))
 if $r_{2s} = 0$ then return (r_{1s}, $\lambda v.$ Left($f_{1s}(v)$))
 if $r_{1s} = r_{2s}$ then return (r_{1s}, $\lambda v.$ Left($f_{1s}(v)$))
 return($r_{1s}+r_{2s}$,$f_{alt}(f_{1s},f_{2s})$)

This half of the function handles the alternative simplification. It uses the following basic simplification rules: $0 + r \rightarrow r$, $r + 0 \rightarrow r$, $r + r \rightarrow r$. First, the two regular expressions that make up the alternative statement are used in a recursive call. This is because there is the potential for each component of the statement to be simplified further. The simplification rules are then checked against the result of the recursive calls. If the result of simplifying r1 is an empty language regular expression then the simplification of r2 is returned. If the simplification of r2 is the empty language regular expression then the simplification of r1 is returned. If the two simplified regular expressions are the same, the simplification of r1 is returned is returned due to the POSIX matching strategy [19]. In any other case, return the two simplified regular expressions as an alternative statement. At each return statement, a rectification function is also returned with the regular expression. This allows a value for the original regular expression to be calculated. Rectification functions used in the alternative simplification are defined as follows [20]:

λv. Right(f2s(v))
λv. Left(f1s (v))

falt(f1, f2) =
    λv. case v = Left(v′): return Left(f1(v′))
        case v = Right(v′): return Right(f2(v′))

At each simplification step, a rectification function is returned that is later used to build a value. It must be considered that simplification may have occurred inside the left and right regular expressions when returning the rectification function. In the case r1 matches the empty language regular expression, it must be the case that r2 matches the given input, therefore a value of Right would be given. As a result, a rectification function that builds a Right value is returned for this level of simplification. However r2 itself may have been simplified, therefore the rectification function returned by the simplification of r2 is given to the right rectification function. This is the same case as if r2 matched the empty language regular expression. A rectification function that builds a Left value is returned with the rectification function returned by simplifying r1. In the case that the simplified regular expressions are the same, the POSIX matching rules state the left regular expression is to match the input [19]. This leads to a rectification function that builds a Left value being returned, again with the rectification function returned from simplifying r1. In any other case no more simplifications can be applied. As simplification may have occurred inside the left and right regular expressions, the rectification function returned with the alternative statement depends on the rectification functions returned from the simplification of the left and right expressions. If a value of Left is rectified then the left rectification function is returned, if a Right value is rectified then the Right rectification function is returned.

$simp(r_1 \cdot r_2)$:
    $(r_{1s}, f_{1s}) = simp(r_1)$
    $(r_{2s}, f_{2s}) = simp(r_2)$
    if $r_{1s} = 0$ then return $(0, f_{error})$
    if $r_{2s} = 0$ then return $(0, f_{error})$
    if $r_{1s} = 1$ then return $(r_{2s}, \lambda v.\ Seq(f_{1s}(Empty), f_{2s}(v)))$
    if $r_{2s} = 1$ then return $(r_{1s}, \lambda v.\ Seq(f_{1s}(v), f_{2s}(Empty)))$
    return $(r_{1s} \cdot r_{2s}, fseq(f_{1s}, f_{2s}))$

This half of the simplification function handles the sequence simplification. It follows the following basic simplification rules: $0 \cdot r \rightarrow 0$, $r \cdot 0 \rightarrow 0$, $1 \cdot r \rightarrow r$, $r \cdot 1 \rightarrow r$. Again the first the two regular expressions that make up the sequence statement are used in a recursive call. This is because there

is the potential for each component of the statement to be simplified further. The simplification rules are then checked against the results of the recursive calls. If the result of simplifying r1 is an empty language regular expression, the empty language regular expression is returned. This means the given input can't be matched by the given regular expression. If the result of simplifying r2 is the empty language regular expression, the empty language regular expression is returned. Again, this means the given input can't be matched by the given regular expression. If the result of simplifying r1 is an empty string regular expression, the simplification of r2 is returned. If the simplification of r2 is the empty string regular expression, the simplification of r1 is returned. In any other case, the two simplified regular expressions are returned as a sequence statement. Similarly to the alternative simplification, each of these return statements return a rectification function along with the regular expression. Again allow a value for the original regular expression to be calculated. The rectification functions used in the sequence simplification are defined as follows [20]:

$$\lambda v.\ Seq(\ f_1(v_1),\ f_2(v_2))$$

$$f_{seq}(f_1, f_2) =$$
$$\lambda v.\ \text{case } v = Seq(v_1, v_2)\text{: return } Seq(\ f_1(v_1),\ f_2(v_2))$$

If r1 is simplified to the empty language regular expression then the given regular expression is unable to match the input. As a result no rectification is required. This is represented by the f_{error}. This is the same, for the case where r2 is simplified to the empty language regular expression. If r1 simplifies to the empty string regular expression then the input is matched by r2. As a result, a rectification function that builds a Seq value is returned, where the first element is a rectification function that builds the empty value and the second element is a rectification function obtained from the simplification of r2. This is the same case for if r2 simplifies to the empty string regular expression, except the rectification function returned builds a Seq value where the second argument is a rectification function that builds the empty value and the first argument is a rectification function obtained from the simplification of r1. In any other case, the returned rectification function is the sequence rectification function with its two components being the two rectification functions from the simplification of the two regular expressions. Lastly if no simplification can be performed for example if the regular expression 'a' is given. Then the given regular expression is returned, no rectification is required. The full simplification function is defined below [20]:

$simp(r)$:
  case $r = r_1 + r_2$
    let $(r_{1s}, f_{1s}) = simp(r_1)$
      $(r_{2s}, f_{2s}) = simp(r_2)$
    case $r_{1s} = \mathbf{0}$: return $(r_{2s}, \lambda v.\ Right(f_{2s}(v)))$
    case $r_{2s} = \mathbf{0}$: return $(r_{1s}, \lambda v.\ Left(f_{1s}(v)))$
    case $r_{1s} = r_{2s}$: return $(r_{1s}, \lambda v.\ Left(f_{1s}(v)))$
    otherwise: return $(r_{1s} + r_{2s}, f_{alt}(f_{1s}, f_{2s}))$
  case $r = r_1 \cdot r_2$
    let $(r_{1s}, f_{1s}) = simp(r_1)$
    $(r_{2s}, f_{2s}) = simp(r_2)$
    case $r_{1s} = \mathbf{0}$: return $(\mathbf{0}, f_{error})$
    case $r_{2s} = \mathbf{0}$: return $(\mathbf{0}, f_{error})$
    case $r_{1s} = \mathbf{1}$: return $(r_{2s}, \lambda v.\ Seq(f_{1s}(Empty), f_{2s}(v)))$
    case $r_{2s} = \mathbf{1}$: return $(r_{1s}, \lambda v.\ Seq(f_{1s}(v), f_{2s}(Empty)))$
    otherwise: return $(r_{1s} \cdot r_{2s}, f_{seq}(f_{1s}, f_{2s}))$
  otherwise:
    return $(r, \lambda v.\ v)$

To add this simplification function to the existing lexer a small modification to the lex function is made. It is defined the same as before but with the introduction of the simp1 function [20].

```
simp_lex r []  = if nullable(r) then mkeps(r)
                 else error
simp_lex r c::s = let (r_s, f_r) = simp1 der(c, r)
                  inj r c f_r(simp_lex(r_s, s))
```

The first statement is unaltered. In the second statement the derivative of r is built with c as the given character. The derivative is then simplified using simp1. Lexing is continued with the simplified derivative regular expression and the rest of the string s. As a result the simp function is applied in a nested manner to all derivative regular expressions. After the entire string has been used to calculate a simplified derivative regular expression, the first statement calls mkeps, thus returning the first value used in the rectification and injection. The returned values are rectified using f_r from the simplification as described above. The results are injected into the value.

3.2.2 Extended regular expressions

The extended set of regular expressions, as described in the introduction to this report, are additional regular expressions that don't give any additional descriptive power. They do provide a more convenient and concise way to define regular languages. As a result, the size of some regular expressions can be smaller if represented using the extended regular expression set. All extended regular expressions can be implemented using just the basic regular expressions.

The set of extended regular expressions are defined as follows [18]:

$$r := [\ char\ ] \mid r+ \mid r? \mid r\{N\} \mid r\{n,m\}$$

Where a description of there functionality is as follows:

[char] = is a choice of a character from a given set.

e.g. [‘a’,’b’,’c’] could match a or b or c. It is equivalent to writing ‘a’+’b’+’c’

r+ = one or more occurrences of r.

e.g. (‘a’)+ would match one or more ‘a’. It is equivalent to ‘a’+(’a’*)

r? = zero or one occurrence of r.

e.g. ‘a’? would match the empty string or ‘a’. It is equivalent to ONE+’a’

r{N} = Exactly N occurrences of r

e.g. (‘a’){5} would match “aaaaa”. It is equivalent to writing ‘a’·’a’·’a’·’a’·’a’

r{n,m} = n to m occurrences of r

e.g. (‘a’){1,4} would match “a” or “aa” or “aaa”. It is equivalent to writing

‘a’ + (‘a’·’a’) + (‘a’·’a’·’a’)

A definition of the language these regular expressions represent is given in the introduction section to this report. Due to the functional definition of derivatives, implementing the extended regular expressions to the set already implemented is simple. Additional rules need to be added to the nullable function, der function, mkeps function and inj function. Adding the extended rules to each function follows the same ideology as that of the basic regular expression, meaning the property of each function does not change. The following are the amended definitions:

nullable (0) = False

nullable (1) = True

nullable (c) = False

nullable (r1 + r2) = nullable r1 ∨ nullable r2

nullable (r1 · r2) = nullable r1 ∧ nullable r2

nullable (r ⋆) = True

nullable [s] = False

nullable r+ = nullable r

nullable r? = True

nullable r{N} = if (n == 0) True else nullable r

nullable r{n, m} = if (n == 0) True else nullable r

der (c, 0) = 0

der (c, 1) = 0

der (c, c) = if c = d then 1 else 0

der (c, (r1 + r2)) = der(c r1) + der(c r2)

der (c, (r1 · r2)) = if nullable(r1)

then (der(c r1) · r2) + der(c r2)

else der(c r1) · r2

der (c, (r⋆)) = der(c r) · (r⋆)

der (c, [s]) = if(s.contains(c)) 1 else 0

der (c, r+) = der(c,r) · r*

der (c, r?) = der(c,r)

der (c, r{N}) = if (n == 0) 0 else der(c, r) · r{n - 1}

der (c, r{n, m}) = if(m <= 0) then 0

else if(n == 0) then der(c,r) · r{n,m-1}

else der(c,r) · r{n-1,m-1}

```
mkeps (1) = Empty
mkeps (r1 · r2) = Seq (mkeps r1) (mkeps r2)
mkeps (r1 + r2) = if nullable r1 then Left (mkeps r1) else Right (mkeps r2)
mkeps (r ⋆ ) = Stars list()
mkeps r+ = Stars list(mkeps(r))
mkeps r? = mkeps(ALT(ONE,r))
mkeps r{N} = Stars((for(g <- 0 until n) yield mkeps(r)).toList)


inj d c () = Char d
inj (r1 + r2) c (Left v1) = Left (inj r1 c v1)
inj (r1 + r2) c (Right v2) = Right (inj r2 c v2)
inj (r1 · r2) c (Seq v1 v2) = Seq (inj r1 c v1) v2
inj (r1 · r2) c (Left (Seq v1 v2)) = Seq (inj r1 c v1) v2
inj (r1 · r2) c (Right v2)  = Seq (mkeps r1) (inj r2 c v2)
inj (r ⋆ ) c (Seq v (Stars vs)) = Stars (inj r c v :: vs)
inj r{N} c Sequ(v1,Stars(vs)) = Stars(inj(r, c, v1)::vs)
inj r+ c Sequ(v1, Stars(vs)) = Stars(inj(r, c, v1)::vs)
inj [s] c Empty = Chr(c)
inj r? c v = Left(inj(r,c,v))
```

The amendments to the functions highlight how easy it is to add and subtract regular expressions. These extended functions are able to operate in conjunction with the simp1 function. There is no need to alter the simplification function as simplification is based on alternative and sequential regular expressions.

3.2.3 Simplification function 2

After performing some matching tests, it was observed that the simp1 function [2] defined earlier in this report is unable to optimise this derivative base lexer for alternative heavy regular expressions. This is because of duplicated statements inside the alternative regular expression. These tests can been seen in the test chapter of this report. As a result, a modification to the simp1 function was made. Simp2 is a simplification function designed by Dr Christian Urban of Kings Collage London and is the result of the modification of simp1 [21]. Simp2 optimises the lexer in a similar way to simp1. The simplification is based on the given simplification rules defined in simp1 [2], these being for alternative and sequence regular expressions. The difference is that simp2 uses an extra argument called '*seen*' to filter out the duplicates. *Seen* is a set of regular expressions. Every time a statement is made it is 'remembered' by storing it in *seen*. If during the simplification, a statement is identified as duplicated, this duplication is removed rather than simplified. This leads to the reduction in the size of the derivative regular expression. The Simp2 function created by Dr Christian Urban of Kings Collage London is defined as follows [21], where ++ is list concatenation:

simp(*r, seen*):
  case $r = r_1 + r_2$
    $(r_{1s}, \text{seen1}, f_{1s}) = simp(r_1, \text{seen})$
    $(r_{2s}, \text{seen2}, f_{2s}) = simp(r_2, \text{seen1})$
    case $r_{1s} = 0$: return $(r_{2s}, \text{seen2}, \lambda v.\ Right(f_{2s}(v)))$
    case $r_{2s} = 0$: return $(r_{1s}, \text{seen2}, \lambda v.\ Left(f_{1s}(v)))$
    otherwise: return $(r_{1s} + r_{2s}, \text{seen2}, f_{alt}(f_{1s}, f_{2s}))$
  case $r = r_1 \cdot r_2$
    $(r_{1s}, \_, f_{1s}) = simp(r_1, [])$
    $(r_{2s}, \_, f_{2s}) = simp(r_2, [])$
    if (SEQ(r1s, r2s)) ∈ seen) return (0, seen, F_ERROR)
    case $r_{1s} = 0$: return $(0, \text{seen}, f_{error})$
    case $r_{2s} = 0$: return $(0, \text{seen}, f_{error})$
    case $r_{1s} = 1$: return $(r_{2s}, \text{seen} ++ \text{r2s}, \lambda v.\ Seq(f_{1s}(Empty), f_{2s}(v)))$
    case $r_{2s} = 1$: return $(r_{1s}, \text{seen} ++ \text{r1s}, \lambda v.\ Seq(f_{1s}(v), f_{2s}(Empty)))$
    otherwise: return $(r_{1s} \cdot r_{2s}, \text{seen} ++ r_{1s} \cdot r_{2s}), f_{seq}(f_{1s}, f_{2s}))$
  otherwise:
    if (r ∈ seen) (0, seen, F_ERROR) else (r, seen ++ r, F_ID) [21]

As with simp1, simp2 still implements rectification functions due to the fact that a value needs to be built for the original regular expression, not the simplified regular expression. The rectification functions are defined and implemented in the same way as in simp1. Simp2 interacts with the other functions the same as simp1, therefore if implementing simp2, all instances of simp1 in the simp_lex function should be changes to simp2.

3.2.4 Bit-code optimisation

Incremental Bit-Coded Forward Parse Tree Construction is another way we can look at optimising the lexer. This optimisation was proposed by Martin Sulzmann and Kenny Zhuo Ming Lu in their paper POSIX Regular Expression Parsing with Derivatives [13] and is further researched in papers such as Certified bit-coded regular expression parsing by Rodrigo Ribeiro [12]. This Bit-Coded parse tree construction tries to optimise the lexer in two ways. Firstly bit-codes are used to represent values more concisely. Secondly, during the matching phase a value is incrementally built up, thus the injection function previously used is no longer required, as it is implemented during the matching. As a result there is no need to record an entire path of derivative expression. To note, the extended regular expressions and the simplification functions are not implemented for the bit-code optimisation in this project.

It is important to understand that this optimisation follows the implementation of the lexer defined in prior sections, such that the algorithm is based around the use of derivatives and follows the POSIX matching rules [19]. The injection function is taken out and implemented in the matching phase, thus a few functions have to be introduced and modified to enable this functionality.

A bit-code represented as ‘b’ is a number 0 or 1. A bit-code sequence represented as ‘bs’ is a list of bit-codes where ‘[]’ represents an empty bit-code sequence. These bit-codes are used to help reconstruct a value. More formally they are defined as follows [13]:

b = 0 | 1
bs = [] | b : bs

To compute a bit-code representation of a value, a function called encode is defined. To compute a value from a bit-code representation, a function called decode is used. Encode is not required for the implementation of the lexer [13] however is useful to implement, as it can be used to check if the result of converting a value to a bit-code sequence is the same as the result of the bit-code algorithm. Decode takes as arguments a regular expression, a bit-code sequence and outputs a value. Encode and decode are defined as follows [13]:

```
encode Empty = []
encode Chr(c) = []
encode (Left v) = 0 : encode(v)
encode (Right v) = 1 : encode(v)
encode (v1 , v2 ) = encode(v1) ++ encode(v2)
encode Stars [] = [1]
encode Stars (v : vs) = (0 : encode(v)) ++ encode(vs)

decode 1 bs = (Empty, bs)
decode c bs = (Chr(c), bs)
decode r1+r2 (0 : bs) = let (v,p) = decoder(r1 bs)
                        (Left v,p)
decode r1+r2 (1 : bs) = let (v,p) = decode(r2 bs)
                        (Right v,p)
decode r1·r2 bs = let (v1,p1) = decode(r1 bs)
                      (v2, p2) = decode(r2 p1)
                     ((v1, v2), p2)
decode r* (0 : bs) = let (v,p1) =decode(r bs)
                         (vs, p2) = decode(r* p1)
                     (Stars(v : vs),p2)
decode r* (1 : bs) = (Empty, bs)

decode' r bs = let (v,p) = decode(r bs)
                   in case p of
                         Empty -> v
```

Bit-codes give information about if an alternative regular expression matched the left or right value and if the star regular expression matched the empty string or not. The star expression matches the empty string in two cases - the input given is the empty string or all of the input is matched and only the empty string is left. If the empty string regular expression is given, this means the empty string was matched. As a result, the value associated with the empty string is returned. As bit-codes have no effect on decoding the empty regular expression, the bit-code sequence is returned without alteration. This is the same case for the character regular expression, except a character value is returned. An alternative statement (r1+r2) can match the given input with r1 or r2, to know which, bit-codes would have been encoded. If a bit-code of 1 is at the front of the bit-code sequence then the r2 matched the input, if it was 0 then r1 matched the input. The regular expression that matched the input is then used in a recursive call. The recursive call is given the original bit-code sequence minus the first bit-code that was used to tell if the match took place in r1or r2. All regular expressions in the sequence regular expression (r1·r2) can match parts of an input, therefore each regular expression in the sequence needs to be used in a recursive call. The original bit-code

sequence is used in the first recursive call, the resulting bit-code sequence of the first recursive call is used in the second recursive call. The star regular expression can match the empty string or a repetition of the language given by the regular expression. To denote this, bit-codes would have been encoded. If a bit-code of 0 is given, like in the sequence case, a recursive call is made on a single occurrence of r* with the original bit-code sequence. The bit-code sequence returned from this call is then used in the second recursive call on r*. If a bit-code of 1 is given, then Empty is returned with the given bit-code sequence. An example is as follows:

Given: r = (a+(b+c)) and an input ‘c’
The value for this is: v = Right(Right(Chr(‘c’)))

Calling encode(v):
Right alternative case is matched: bs = 1 : encode(Right(Chr(‘c’)))
Right alternative case is matched: bs = [1] ++ 1 : encode(Chr(‘c’))
Chr is matched: bs = [1,1] ++ []
The result of encode is thus a list [1,1]

If then calling decode on this input decode(r, [1,1])
alternative case is matched: v, p = decode ((1+c),1)
alternative case is matched: v’,p’ = decode (c,[])
v’ = Chr(c); p’= []
(Right(Chr(c)),[]) is returned
v = Right(Chr(c)); p = []
Right(Right(Chr(c))) is returned
value = Right(Right(Chr(c)))

To construct a value during matching, an ability to store information in the form of a bit-code sequence during the derivative step is required. At current, the defined regular expressions don’t allow for the storing of information in any form. Sulzmann & Lu proposed altering the definition of regular expressions such that they could be ‘annotated’ with information on how a value can be constructed [13]. Annotated regular expressions are regular expressions that can hold information in

the form of a bit-code sequence, and are used in conjunction with regular expressions in the algorithm. Annotated regular expressions are defined as follows [13]:

$$ri = 0 \mid (bs \ @ \ 1) \mid (bs \ @ \ c) \mid (bs \ @ \ ri \oplus ri) \mid (bs \ @ \ ri \cdot ri) \mid (bs \ @ \ ri^*)$$

Each annotated regular expression is obtained from the basic regular expression by adding the ability to store partial parse tree information in the form of bit-code sequences. This is represented by the 'bs'. @ depicts the separation of bit-code sequences from the regular expressions. As the empty language regular expression represents no value, the annotated regular expression of the empty language does not need to hold any data and thus doesn't require the ability to store a bit-code sequence. All the other annotated regular expressions are built by adding empty bit-code sequences. The last difference is that choice (r1 + r2) is redefined to use the ⊕ symbol. This is done to show that all the information to construct a value can be obtained from the operands of ⊕ without having to inspect the surrounding structure.

To be able to annotate annotated regular expressions a function called fuze is introduced. Fuze takes as input a bit-code sequence and an annotated regular expression. The output is an annotated regular expression, where the given bit-code sequence, and the bit-code sequence of the given annotated regular expression, are combined. This function is used as a helper method for many of the functions to be defined. This is because to be able to build up these bit-code representations, an ability to add bit-code information is required. Fuze is defined as follows [13]:

$$\text{fuse bs } 0 = 0$$
$$\text{fuse bs } (p@1) = (bs{+}{+}p@1)$$
$$\text{fuse bs } (p@c) = (bs{+}{+}p@c)$$
$$\text{fuse bs } (p@ri1 \oplus ri2) = (bs{+}{+}p@ri1 \oplus ri2)$$
$$\text{fuse bs } (p@ri1 \cdot ri2) = (bs{+}{+}p@ri1\ ri2)$$
$$\text{fuse bs } (p@ri^*) = (bs{+}{+}p@ri^*)$$

Fuzing a bit-code sequence with the empty language annotated regular expression, results in just the empty language annotated regular expression, as there is nothing to fuze this given bit-code sequence with. For all the other cases the bit-code sequence of the given regular expression is appended to the end of the given bit-code sequence and then recombined with the annotated regular expression.

Now, there is the problem of how to convert the regular expressions into the annotated regular expressions. This is required as the derivative function will be working with annotated regular expressions. When calling the lexing function, a regular expression is given, not an annotated regular expression. To do this the internalize function is introduced. Internalize converts a regular expression into an annotated regular expression by inserting empty bit-code sequences. Internalize takes as an argument a regular expression and outputs an annotated regular expression, it is defined as follows [13]:

internalize $0 = 0$

internalize $1 = ([]@1)$

internalize $c = ([]@c)$

internalize $(r1 + r2) = ([]@(\text{fuse } [0] \text{ (internalize } r1)) \oplus (\text{fuse } [1] \text{ (internalize } r2)))$

internalize $(r1 \cdot r2) = ([]@(\text{internalize } r1) \text{ (internalize } r2))$

internalize $r* = ([]@(\text{internalize } r)*)$

The empty language does not represent any value and therefore doesn't need to store any information, so the annotated regular expression of the empty language is the same as the regular expression. The empty string and character expressions are combined with an empty bit-code sequence to obtain the annotated regular expression. The alternative (r1 + r2) is converted to the $\oplus$ symbol and the bit-code 0 is fuzed to the result of the recursive call of internalize on r1. The same is done to r2 but a bit-code of 1 is fuzed to the result of the recursive call. The sequence regular expression $(r1 \cdot r2)$ is combined with an empty bit-code sequence, however due to r1 and r2 being regular expressions not annotated regular expressions, they must also be internalized into annotated regular expressions, hence the internalize call on r1 and r2. The star regular expression is combined with an empty bit-code sequence and as the regular expression inside the star is not a annotated regular expression internalize is called.

An example of the operation of internalize is as follows:

given a regular expression r = (a+(b+c)) and ':' is the binary operator fuze

internalize(r) = [] @ 0 : internalize(a) ⊕ 1 : internalize(b+c)

= [] @ 0 : ([]@a) ⊕ 1 : ([]@0:internalize(b) ⊕ 1:internalize(c))

= [] @ 0 : ([]@a) ⊕ 1: ([]@ 0:[]@b ⊕ 1:[]@c)

= [] @ ([0]@a) ⊕ 1 : ([]@ [0]@b ⊕ [1]@c)

= [] @ ([0]@a) ⊕ ([1]@ [0]@b ⊕ (1]@c)

At current, functions have been defined that allow for describing bit-code sequences, describing annotated regular expressions, allowing for the conversion of bit-codes into values and back, the combining of bit-code sequences with annotated regular expressions and finally a function that obtains an annotated regular expressions from regular expressions. All that needs to be done now is to redefine the derivative function. The derivative functioned needs to operate on annotated regular expression to build up a bit-code sequence. The new derivative function that operates on annotated regular expressions is called Ider. The name is changed from der to Ider to avoid confusion and to be able to distinguish the two functions. Ider operates in a similar way to der. The function takes as arguments a character, an annotated regular expression and outputs a new annotated regular expression. Assuming the given annotated regular expression can match a string of the form c :: s, the new annotated regular expression can match a string of the form s. In addition to this it copies and inserts value information in terms of bit-codes. It can be thought that Ider is a combination of both the der function and the inj function from the original algorithm. Ider is defined as follows [13]:

Ider c' 0 = 0

Ider c' (bs@1) = 0

Ider c' (bs@c) = if c' == c then (bs@1)

else 0

Ider c' (bs@ri1 ⊕ ri2) = (bs@ Ider(c', ri1) ⊕ Ider(c', ri2))

Ider c' (bs@ri1 · ri2) = if nullable(ri1)

then bs@ ([]@Ider(c',ri1) ri2) ⊕ (fuse(mkEpsBC(ri1), Ider(c,ri2)))

else bs@Ider(c', ri1) ri2

Ider c' (bs@ri∗) = (bs@(fuse [0] Ider(c', ri)) ([]@ri∗))

Calling Ider on the empty language annotated regular expression, returns just the empty language annotated regular expression as no string of the form c' :: s can match this. This is the same for the empty string annotated regular expression. Calling Ider on the character annotated regular expression is similar to that of the der function. If c' can be matched by the annotated regular expression an annotated regular expression that matches a string of the form s is returned, if the given string is of the form c' :: s. As the given annotated regular expression can only match characters, the result must be an annotated regular expression that matches the empty string, thus returns the empty string annotated regular expression. If it can't match c' then the empty language annotated regular expression is returned. No annotations are made to the bit-codes at this point, this is because no annotation is required if just matching a character. For the alternative expression (bs@ri1 ⊕ ri2), either the ri1 or ri2 matched the input, therefore Ider is called on both ri1 and ri2. No annotation to the expressions are made here as the bit-codes were added when calling *internalize*. Like in der, when considering the sequence case (bs@ri1 · ri2) it must be taken into account that ri1 can match the empty string and thus c' can be matched by ri2. To tell if an annotated regular expression matches the empty string a small alteration to nullable is made, all regular expressions are changed to annotated regular expressions [13].

nullable 0 = false

nullable (bs@1) = true

nullable (bs@c) = false

nullable (bs@ri1 ⊕ ri2) = nullable(ri1) || nullable(ri2)

nullable (bs@ri1 · ri2) = nullable(ri1) && nullable(ri2)

nullable (bs@ri∗) = true

Continuing with the description of the sequence case in Ider. If the empty string can't be matched by ri1, then ri1 must match c'. As a result, the sequence annotated regular expression is returned where a recursive call of Ider is made on ri1 with c'. If it is the case where ri1 matches the empty string, ri1 or ri2 matches c'. This leads to the returning of an alternative annotated regular expression where the first component is the same as if only ri1 matched the input, and the second component considers the case where ri1 matched the empty string, therefore calling mkepsBC on ri1 and fuzing it with a recursive call of Ider on ri2 and c'. mkEpsBC operates in a similar fashion

to that of mkeps. Given an annotated regular expression it returns an annotated bit-code sequence of how the given expression matches the empty string [13].

mkepsBC(bs@1) = bs

mkepsBC(bs@ri1⊕ri2)

| nullable(ri1) = bs++mkepsBC ri1

| nullable(ri2) = bs++mkepsBC ri2

mkepsBC(bs@ri1 · ri2) = bs++mkepsBC ri1 ++mkepsBC ri2

mkepsBC(bs@ri∗) = bs++[1]

The star annotated regular expression must record the number of iterations it has performed ri. This is done by fuzing a bit-code of 0 in each iteration it doesn't match the empty string. If it does match the empty string, a bit-code of 0 is fuzed with the empty string annotated regular expression therefore nothing is fuzed. Calling Ider on the star annotated regular expression returns a sequence regular expression where the first component is a recursive call on a single occurrence of ri* (the result of this is fuzed with 0) and the second component is r*, this allows ri* to match multiple occurrences of ri. An example of Ider operating on the annotated regular expression obtained from internalize(a+(b+c)) as calculated prior is as follows:

Ider([] @ ([0]@a) ⊕ ([1]@ [0]@b ⊕ (1]@c))

= [] @ Ider(c,[0]@a) ⊕ Ider(c, [1]@([0]@b)⊕([1]@c))

= [] @ 0 ⊕ [1]@Ider(c,[0]@b) ⊕ Ider(c,[1]@c)

= [] @ 0 ⊕ ([1]@0 ⊕ [1]@1)

An then using mkeps to get the bitcode sequence

mkepsBC([] @ 0 ⊕ ([1]@0 ⊕ [1]@1))

= [] ++ mkepsBC([1]@0 ⊕ [1]@1)

= [] ++ [1] ++ mkepsBC([1]@1)

= [] ++ [1] ++ [1]

= [1,1]

The result of mkepsBC is [1,1]. This the same result that was obtained earlier when using encode to calculate a bit-code sequence for the corresponding value. As shown earlier the decode function would then produce the expected value from this bit-code sequence.

Now all components have been redefined and or introduced we can redefine the lex function [13].

```
lex ri []    = if nullable(ri) then mkepsBC(ri)
               else error
lex ri c::s = lex(Ider(c, ri)), s)

lexer r s = decode(r, lex(internalize(r), s.toList))
```

lexer first converted the given regular expression into an annotated regular expression using internalize. lex then checks if the given annotated regular expression matches the empty string. If it can, mkepsBC is used to obtain the bit-code sequence for how this annotated regular expression matched the empty string. If the given annotate regular expression does not match the empty string, lex iterates the derivative function with the characters of the string. Given the expression can match the input, the result of iterating Ider is an annotated regular expression that matches the empty string. The resulting bit-code sequence is then decoded thus producing a value for how the regular expression matched the given input.

Chapter 4

Testing

4 optimisations have been researched and analysed in this report, these being the simp1 function, extended regular expression set, simp2 and bit-code parse tree construction. The simplification functions both concentrated on reducing the size of the derivative regular expression being built in the matching phase. The extended regular expressions optimised by providing a way of concisely defining the regular expression in the first place, for example, a{30} is one expression if using the extended regular expressions but is 29 when using the basic regular expressions. Bit-code forward parse tree construction optimised by building a value at the same time as calculating the derivative regular expression and concisely representing these values using bit-codes.

These optimisations are trying to optimise this lexing algorithm [1] in the sense of time and memory. They try to reduce the size of the regular expression. As a result less memory is required to store the regular expression. The effect this has is that now there are less expressions to analyse when lexing an input. This leads to a decrease in the time it takes to lex an input.

One of the major concerns with lexical analysis and matching in general is the effects of 'evil' regular expressions on the time it takes. The experiments here are looking into what evil regular expressions the optimisations allow the lexer to handle and what they are unable to allow it to handle. First the matcher is tested. For convenience, the alternative regular expression (r + r) is redefined as (r | r) to avoid confusion with the plus extended regular expression.

4.1 Matcher testing

Python, Java and Ruby are introduced to provide benchmark times to compare the implemented matchers against. Basic, simp1 and simp2 in the graphs all represent Brzozowski matcher [3] using the given optimisation, with basic being the matcher with no optimisation.

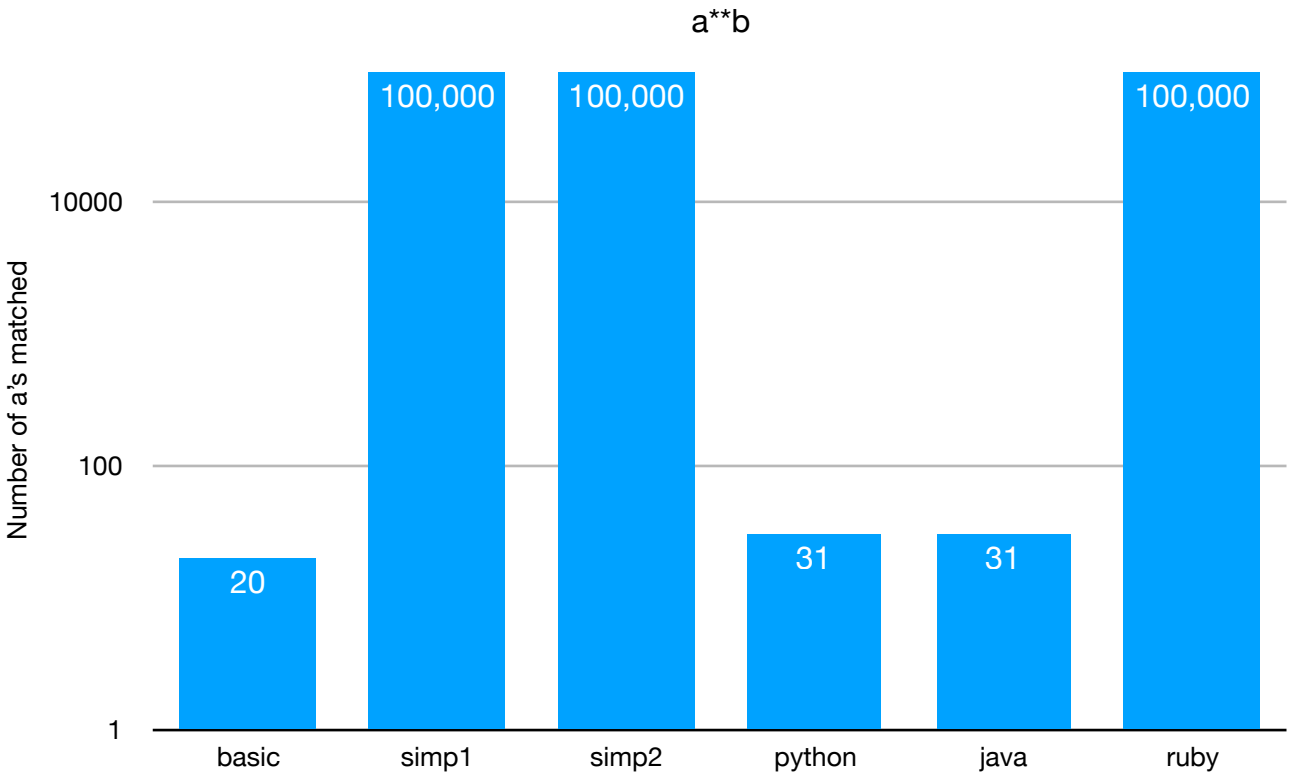

Figure 4.1: Shows the number of a's matched by different optimised matchers [3]

Figure 4.1 shows matching with the evil regular expression a**b. The basic matcher was able to try to match 20 a's in 1.27675s. Scala returned the following message when trying to match 21 a's, 'java.lang.OutOfMemoryError'. The matcher using the simp1 optimisation was able to try to match 100,000 a's in 0.02023s. Inputs of larger size are able to be tried, however 100,000 was the maximum size input tried for this test. Simp2 again was able to match 100,000 a's in 0.03947s. Like with simp1 inputs of larger size are able to be tried to match but 100,000 was the maximum size input tried for this test. Bit-code doesn't implement a matcher therefore no test was performed. Python and Java where both able to match 31 a's in 211.062469006s and 144.993193808s respectively. Matching of larger inputs were not performed in Python and java due to the length of time it took to match. Ruby was able to match 100,000 a's in 0.00410s. Again the max size input for the test was 100,000 however larger inputs are able to be tried to be matched. From this data it can be seen that the simplified matchers are able to match larger inputs in less time than that of Python and Java. At the same time they look to be able to perform at a similar level to ruby. This however is all with respect to the given evil regular expression. The basic matcher matched the smallest input out of all the matching implementations. It was the only matcher to return a memory overflow error. To identify why the basic matcher performed in the way it did, the size of the derivative built was investigated.

Figure 4.2 Shows the size of the derivative regular expressions produced by the basic matcher given the evil regular expression a**b with respect to different input sizes. The basic matcher's derivative size increased exponentially as the input size increased, given an input size of 20 a's the matcher built a derivative regular expression of size 7,000,000+. This gives an explanation as to why a 'java.lang.OutOfMemoryError' was given. Due to the size of the regular expression, it was not able to be stored, this is because not enough memory was allocated. As a result of such a large expression being built, the time in which it took to analyse such an expression also increased exponentially, thus giving reason as to why it took 1.27675s to match 20 a's. In comparison, both simplification functions where able to keep the size of the derivative regular expression constant, at a size of 8. Figure 4.2 only shows the size up to an input size of 20, however the constant size of 8 is true for up to a size of 100,000 a's (This is where the texting size stopped). As a result the simp functions were both able to match far larger inputs in less time, therefore being more efficient in both space and time. This leads to a correlation between the size of the derivative regular expression and the time it takes to match. As the size of the derivative regular expression increases so does the time it takes to match.

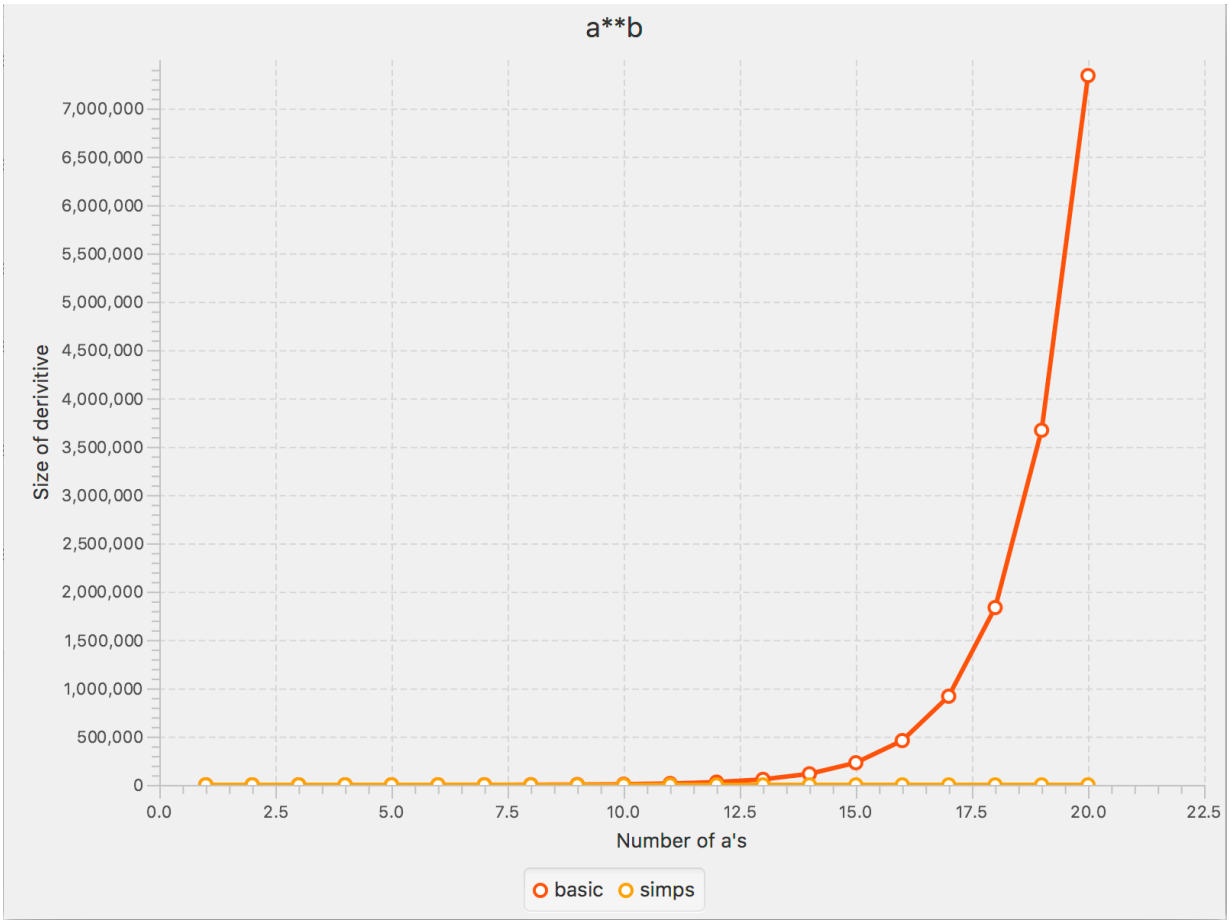

Figure 4.2: Size of derivative regular expression [3]

As stated prior, these results only show how effective the optimisations are with respect to the given regular expression. The simplification functions do not optimise the matcher for all evil regular expressions.

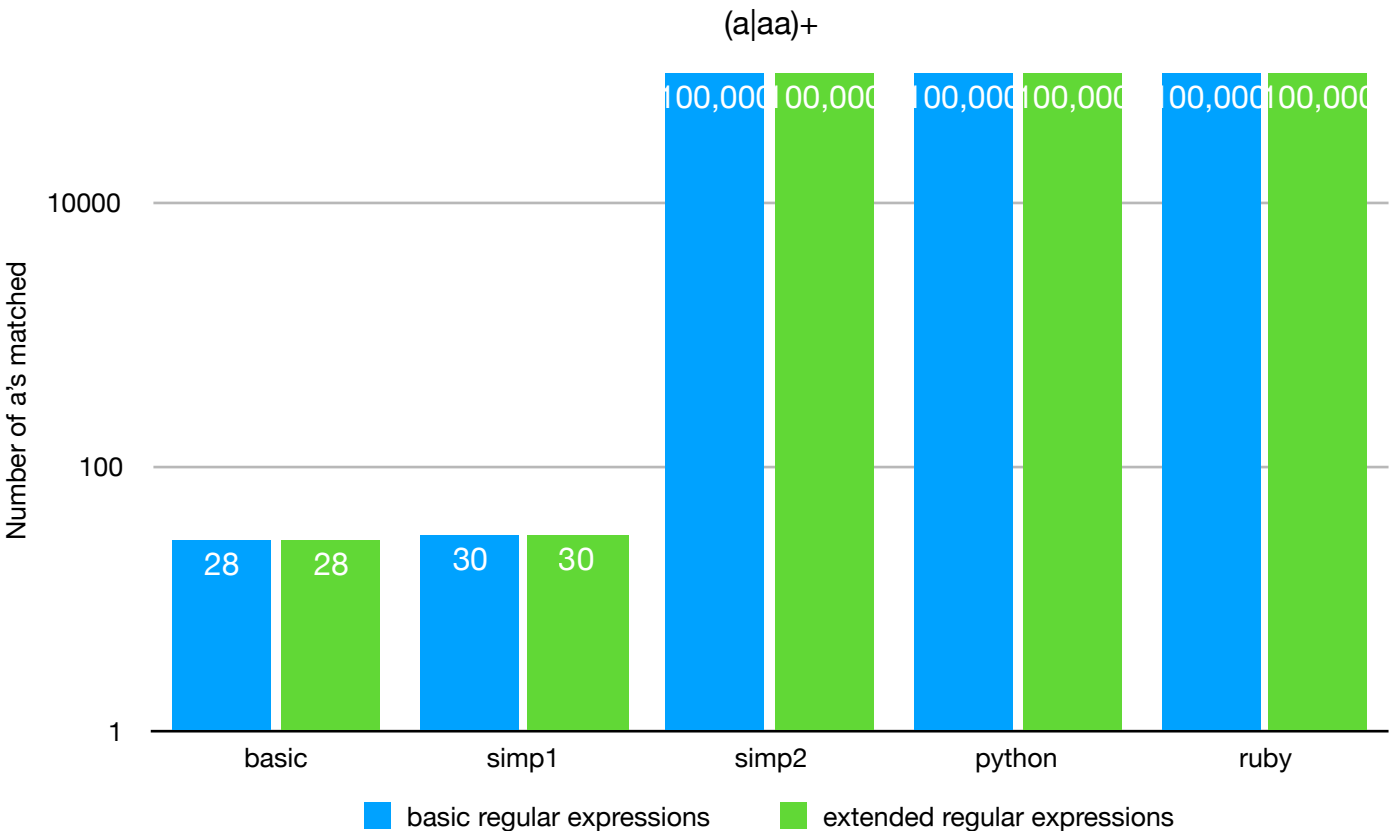

Figure 4.3: Number of a's matched by the different optimised matchers [3] using (a|aa)+

Figure 4.3 shows the matchers trying to match against the evil regular expression (a|aa)(a|aa)* / (a|aa)+ using both the basic and extended regular expression sets. The fist notable point is that simp1 is only able to match 30 a's followed by an out of memory error. It matches 30 a's in 6.07615s when using the basic regular expressions and 30 a's in 8.78593s when using the extended regular expressions. This is an example where the simp1 function is unable to optimise the matcher. The size of the derivative regular expression is 12,752,042 for both basic and extended regular expression implementations when trying to match 30 a's. This therefore shows that the simplification function is has little effect on keeping the derivative regular expression small and at a constant size. This explains why the matcher returns a memory overflow and takes comparatively longer to match. The simp2 matcher however is able to handle this regular expression. It is able to match 100,000 a's in 0.06287s when using the basic regular expressions and 0.04289s when using

the extended set. The size for the derivative for both is 17 and this is constant through out the matching. Python and Ruby are both able to match 100,000 a's in times 0.2s for both the basic and extended regular expression sets. The second notable point from figure 4.3 is that the extended regular expression set had minimal impact on the time and space efficiency of the matchers for this given expression. For the cases where the matchers performed 'badly'; a possible reason for this could be that the alternative statements grow at an exponential rate and as a result no extended regular expression was able to be used to minimise this. Following on from this, in the case of simp1, simplification of the alternative statements is not efficient enough to allow for such an expression to match large inputs. This is one of the main reasons for the creation of simp2 by Dr Christian Urban [21]. However this does not mean the the simp2 function is more efficient than the simp1 function.

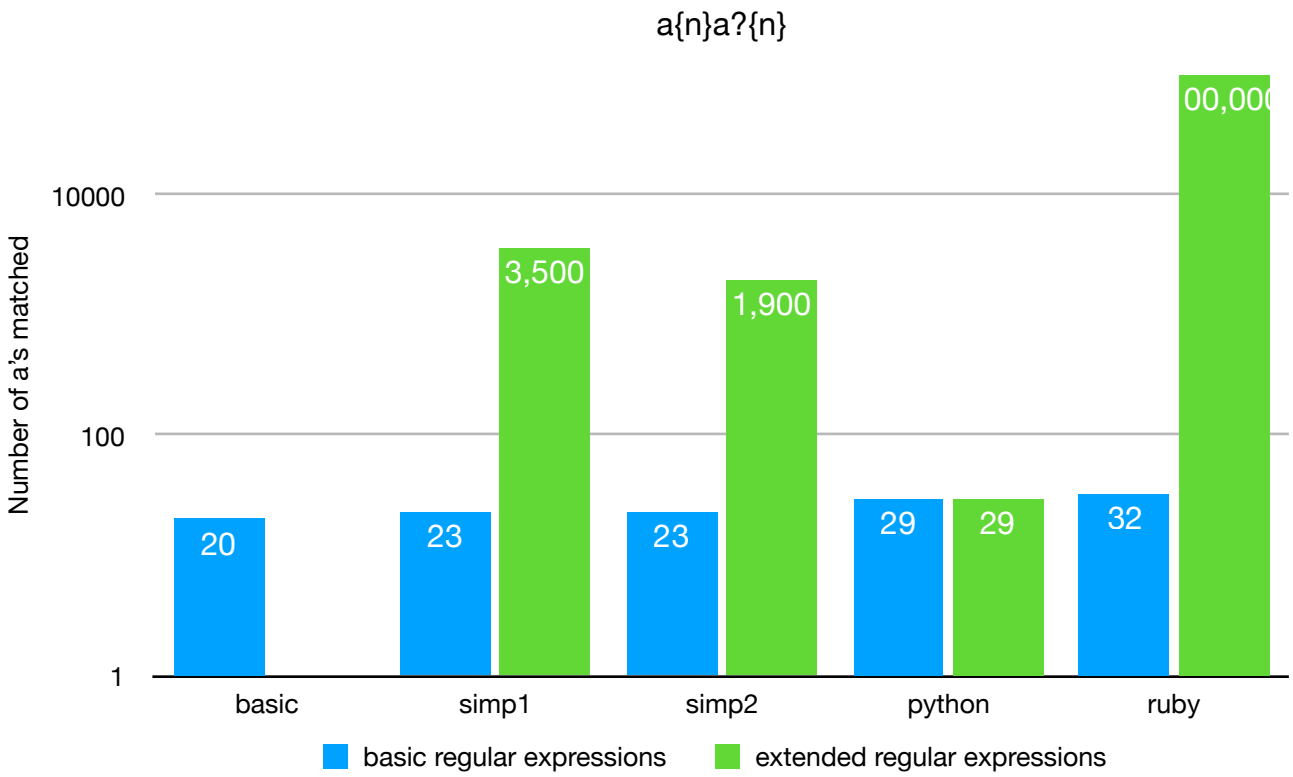

Figure 4.5: Shows the number of a's matched by the different optimised matchers [3]

From figure 4.5 it can be seen that non of the matchers are able to handle the regular expression a{n}a?{n} when using the basic regular expressions. By implementing the extended regular expressions most of the matchers are able to improved the number of a's they are able to match. The basic matcher was able to match 20 a's in 4.77053s using the basic regular expression set. The size of the derivative was 8,394,299, resulting in a memory error when trying to match inputs of greater size. The basic matcher was not tested using the extended regular expressions. Simp1 was able to

match 23 a's in 8.85230s where a memory overflow error was given after this point. The size of the derivative regular expression was 553. In comparison, when simp1 used the extended regular expression set, it was able to match 3,500 a's in 0.34663s. The size of the derivatives regular expression was 10,506. There are a few interesting questions that arise by looking at the results of simp1. When it is matching using the basic regular expressions it is only able to match 23 a's however the size of the derivative regular expression is only 553. This contradicts the previously assumed correlation of size to time. One possible reason for this occurring is that the time it takes the simplification function to operate increases. As a result it may be able to reduce the number of regular expressions being used to build the regular expression resulting in why the derivative is only 553, but at the same time take longer to complete the simplification function. This leads to the overall matching time increasing. Secondly, when matching using the extended regular expression, the size in which the derivative regular expression grew was linear. The size of the derivative grew by 3 for each additional 'a' being matched, where the initial size was 6. In previous tests, either the size grew exponentially or was constant, thus giving reference to whether the optimisation worked for the given evil regular expression or not. This shows that the simp1 function is able to optimise this matcher for this evil regular expression however, for sufficiently large input sizes simp1 would not be able to optimise the matcher well enough to enable it to handle such an input. This leads to the need to find another solution to optimise this matcher for such an evil regular expression if considering larger inputs. However it is sufficient if inputs of less than 3,500 are being used. When looking at the results of simp2, similar occurrences arise. Simp2 matches 23 a's in 13.76830s when using the basic regular expressions. The size of the derivative is 553. Again the reasons for these results are the same as for simp1. Simp2 is able to match 1,900 a's in 0.84593s when using the extended regular expressions. The size of the derivative is 5,706. The same linear growth of 3 in the derivative size occurred during simp2. To note here no errors were returned after matching for simp1 and simp2 when using the extended regular expressions, while testing it was decided to stop matching at these points.

When not using the extended regular expressions, all of the matchers perform at a similar standard with python and ruby being the most efficient. When using the extended regular expressions, simp1 and simp2 are able to optimise their matchers better than that of python, with respect to the regular expression. As a result, the set of evil regular expressions the simp1 and simp2 matchers are able to handle increases by implementing the extended regular expressions. Figure 4.5 also shows an instance where simp1 operates more efficiently than simp2. Simp1 is able to match 3,500 a's in 0.34663's compared to 1,900 a's in 0.84593s for simp2. This shows that different optimisations

perform better for different evil regular expressions. This enforces the fact that each test gives information with respect to the given evil regular expression and doesn't show its efficiency for all evil regular expressions.

4.2 Lexing testing

The optimisations (with the exception of bit-codes) optimise the matching phase of the lexer. The lexing time will therefore be similar to that of matching, but with the value calculation overhead.

Basic, simp1, simp2 and bit-code all represent the lexer proposed by Martin Sulzmann and Kenny Zhuo Ming Lu [1] with the corresponding optimisation applied. The basic lexer has no optimisation. Due to the bit-codes not implementing the extended regular expression set, only the basic regular expressions are used in these tests. Memory was increased to 3GB resulting in the lexer being able to lex more a's in some instances compared to the matcher. This was done to see the exponential runtime of the lexer. The following three graphs give an overview of the lexer performance with respect to the three evil regular expressions that were tested against the matcher.

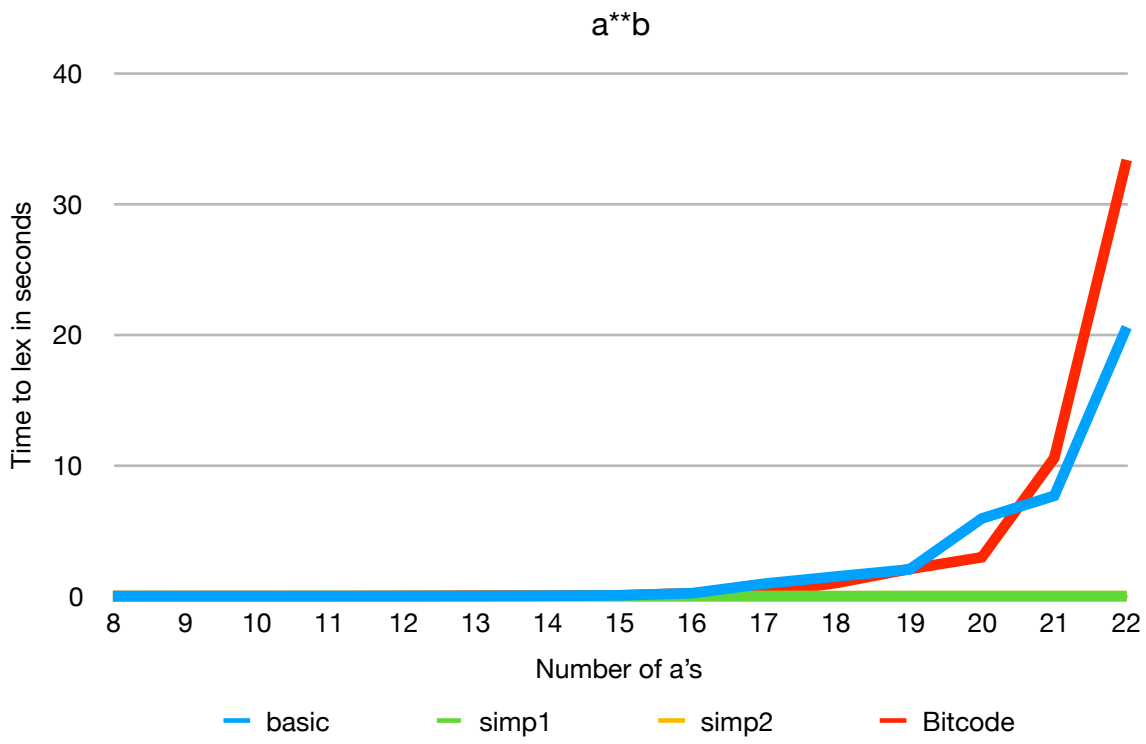

Figure 4.6: Displays the runtime of the different optimised lexers [1] on a**b

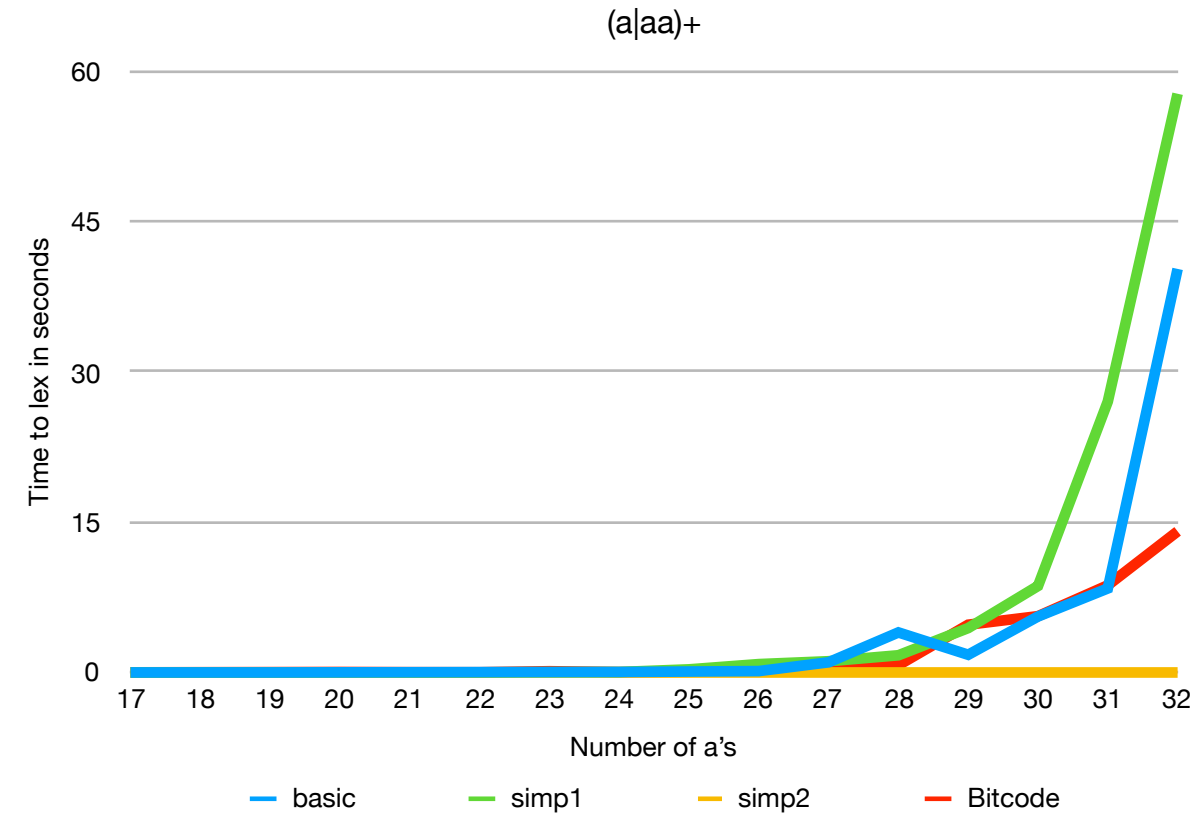

Figure 4.7: Displays the runtime of the different optimised lexers [1] on (a|aa)+

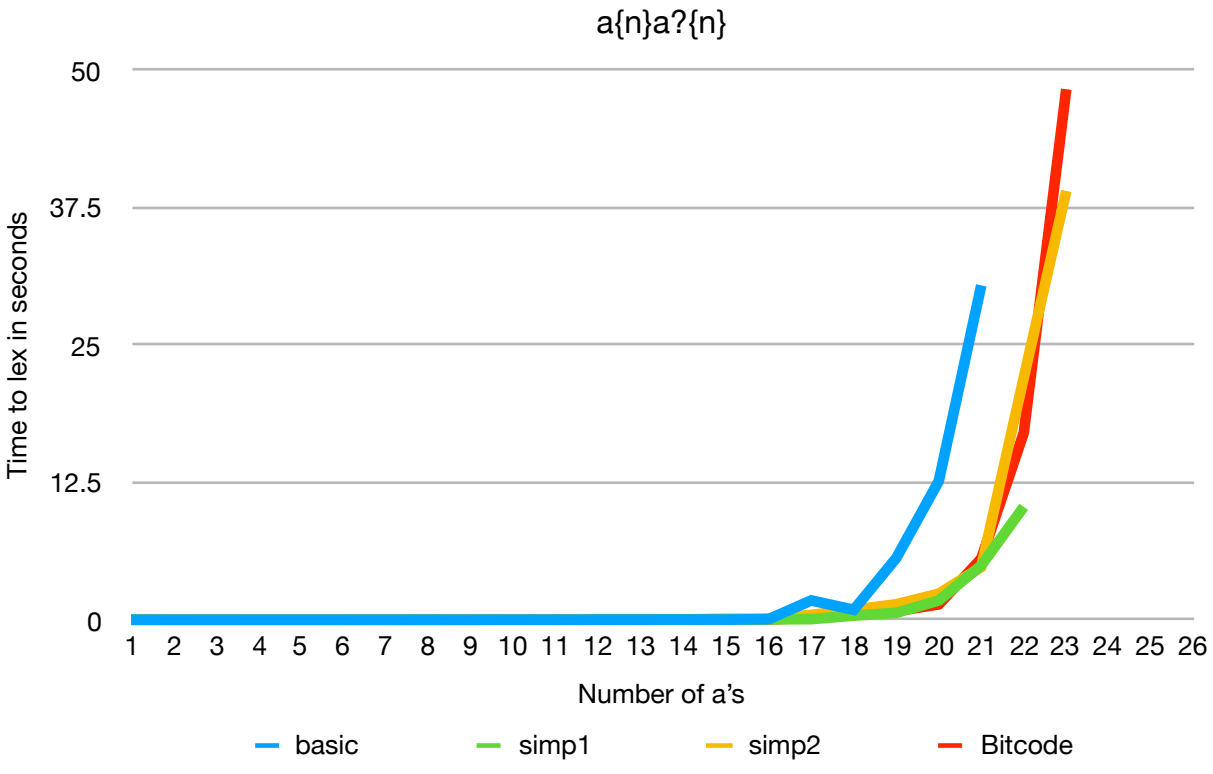

Figure 4.8: Displays the runtime of the different optimised lexers [1] on a{n}a?{n}

The first thing to note about the graphs above is that the run times for basic, simp1 and simp2 is similar to that of there corresponding matcher. This is with exception to simp1 and simp2 when the a**b evil regular expression is used. The lexer must be able to match all parts of the input string. As a result the input string was changed to a sequence of a's followed by a single b as opposed to just a sequence of a's used in the matcher. Therefore the derivative regular expression size became constant at 1 and thus the matcher operated differently to the way it did in the pervious section. The similarities in runtime for the other cases are due to the fact that the optimisation of the lexer occurs in the matching phase. As the value construction is similar for all three optimised variations of the Lexer, there is little change in run times if taking into account the value construction overhead. Due to most of the explanations for the behaviour of these lexers being described in the matching tests, these graphs are mainly for comparison against the bit-code optimisation.

In the case of the a**b regular expression, the bit-code optimisation was able to match 22 a's in 33.42296s with the size of the derivative annotated regular expression being 58,720,298. This grew exponentially throughout the lexing, a memory overflow error was returned when trying to lexer larger inputs. In comparison, the basic lexer was able to match 22 a's in 20.61080s. Simp1 and simp2 were able to match up to 100,000 a's in under 0.01s. The reason for the memory overflow error is similar to that of the cases in the matching. The size of the annotated regular expression was to large to be stored in memory thus memory overflow. Due to the size increasing exponentially so too did the time it then took to lex. The bit-code optimisation was able to lex the same number of a's as the basic lexer, therefore for this evil regular expression, the bit-code optimisation was unable to increase the efficiency of the lexing algorithm.

Looking at the case where the lexers are matching against the evil regular expression $(a|aa)+$, rewritten as $(a|aa) \cdot (a|aa)^*$. The bit-code optimisation enables the input to be lexed in a shorter time than that of the basic and simp1 lexers. The bit-code implementation lexed 32 a's in 14.07713s with the derivative annotated regular expression size reaching 81,186,671. As a result a memory overflow error occurred for inputs of larger size. This is in comparison to the basic and simp1 lexers lexing 32 a's in 40.25687s and 57.72836s respectively. Although the bit-code optimisation performs better than that of the basic lexer or using the simp1 optimisation, matching only 32 a's results in this optimisation not being efficient enough for use in application. This is because most applications of this lexer will require it to be able to handle inputs of greater size than 32.

For the evil regular expression a{n}a?{n} the bit-code optimisation was able to lex 23 a's in 48.30028s. The size of the derivative annotated regular expression was 553. In this test all of the lexers perform poorly, explanations for why simp1 and simp2 perform as such is due to the matching performing poorly and thus described in the prior section.

Many more evil regular expressions were performed on the lexers and matchers as a part of the testing. These three evil regular expressions were chosen to discuss as they best displayed the range of properties shown by the lexers and matchers. These being for example the effect on the size of the derivatives, optimisers working better with different evil regular expressions, the effects of extended regular expressions etc. From the tests that have been presented it can be concluded that the simp1 and simp2 optimisations combined with the extended regular expressions enable the lexer [1] to handle the largest set of evil regular expressions. The bit-code optimisation [13] looks to provide little optimisation to the lexer. It must be taken into account that only a small set of possible evil regular expressions have been tested therefore this conclusion is based only on this small sample. Although the bit-code optimisation looked to perform poorly, there is still additional material that can be applied to the optimisation such as including the extended regular expressions. In the paper by Sulzmann and Lu [1] where the bit-code algorithm is defined [13], a simplification function is also described for the bit-codes. This simplification was not implemented in this project however is a factor to take into account when comparing the performance.

4.3 A simple C lexing grammar

The application of a lexer is not to test how many a's it is able to match, as has been the bases of the experiments so far. As mentioned in the introduction sections they are used in a large variety of text based processes such as syntax highlighting, lexing programs, finding patterns in hostile network traffic, web-crawlers, dictionaries, DNA-data, ad filters etc. When lexing in an application such as compilation, a lexing program will have to tokenise a programming language, for example Scala or Python. A programming language can be broken down into components of which some include keywords, identifiers, integers, floats, strings, operators, whitespaces, comments and many more. A simple C-language lexing grammar was implemented following the lexing rules as defined from Programming languages — C [15]. The C lexing rules where implemented using the derivative regular expressions. Glibc was used as a source of C programs to see how many programs could be lexed using a simple C-lexer. Glibc was downloaded from GitHub with the source directory being

glibc [16]. A pool of 248 C programs where chosen from glibc to lex. The files were taken form the file location glibc/posix/ . Only the simp1 lexer was tested in tokenising these programs. This is because the simp1 lexer was the only implementation that included the record regular expressions. This is with the exception of basic lexer. The simp1 lexer was able to lex 63 and failed 185 C programs, it did this in 7.094653243s . As a result it took on average 0.02861s to lex one C program. Where the lexer was unable to tokenise, a stack overflow error was returned.

Chapter 5

Professional and ethical issues

5.1 British Computing Society Code of Conduct & Code of Good Practice

The Code of Conduct & Code of Good Practice issued by the British Computer Society was strictly followed during all phases of the development of this project. Following the code of conduct protects all parties concerned with such a project, and helps with avoiding legal and ethical issues. Many sources of information in this report are taken from different open source academic research papers. The code of conduct helps protect intellectual property from being illegally used. This for example could be using others research without proper referencing, such an improper use of information is unethical. This project was based around the use of a pre-proposed implementation of a lexing algorithm, therefore all relevant information in this report that is not my personal work has been explicitly stated.

5.2 Packaging for release

Packaging this lexer as a software product would require a few considerations. Firstly, would Scala impose restrictions on the distribution? Secondly the lexer is implemented based on the functional design given by Sulzmann and Lu [1], with optimisations coming from Fahad Ausaf, Roy Dyckhoff, and Christian Urban [2]. As a result, would this pose any restrictions? In terms of Scala posing a distribution barrier; If a copy of the Scala licence is contained within the code and the following three conditions are met as stated on the Scala webpage [14]:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the EPFL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

there would be no legal barrier imposed by Scala. As the lexing algorithm and definitions are in the public domain, the contributors to the respective papers by Martin Sulzmann, Kenny Zhuo Ming Lu, Fahad Ausaf, Roy Dyckhoff, and Christian Urban would not require any monetary compensation to allow for the distribution of the product. There would of course be many other legal cases that would need to be considered to move forward with packaging this lexer for distribution.

Chapter 6

Evaluation

6.1 Project

Background research into how the proposed lexer operated took a large amount of time. Obtaining a good understanding of how derivatives could be used to represent regular expressions, and how they can then be applied to a lexing algorithm was essential. This is because the basic lexer was the foundation the optimisations where built off. Understanding how bit-codes could be used to optimise the lexing algorithm took the most amount of time to research and implement out of all the optimisations. This is because it took a different approach to optimisation when compared to simplification. Although bit-codes build of the original lexer, it required the most amount of code implementation. Along with this, the bit-code research papers where difficult to understand due to the notation changing in each research paper.

The implementation of the functions was easy due to the use of the programming language Scala. This was mainly as a result of Scala's pattern matching ability. Overall a functional language such as Scala was the best fit for the project. If a non functional language such as Java or C was used, the implementation of the algorithm would have required a larger amount of code, and taken more time to implement. This is because Java, C and other alike programming languages don't provide pattern matching. As a result, the functions could not be implemented in the same way. A series of 'if' statements could be used as an alternative to pattern matching, however this would not be an elegant solution. Overall, implementing this algorithm in Java or C would result in the project taking longer, the code would be less concise, harder to read and therefore harder to fix should any problems arise.

The textual notation used by derivatives, combined with the concise patterned matching of Scala, kept the size of the project relatively small. It took 100 lines of code to implement the lexer with no optimisation and 100 extra lines of code to implement simplification and the extended regular

expressions. As a result the derivative based lexer is concise and, once understood, takes relative little effort to implement. An automata implementation would require a greater amount of code and though processing. Questions arise such as how a transition should be represented, how states are stores etc. This leads to the fact that the implementation of a data structure is a code intensive task. When comparing automata to derivative based implementations, derivatives are quicker and more concise to implement.

6.2 Tests

The test approach was done in a well structured manner. The tests focused on matching. This was done as the majority of the optimisations effected the matching phase of the algorithm. As a result, more detail data was obtained as to what the optimisations where doing.
The tests covered a large range of evil regular expressions for which the lexer had to match inputs to. In all instances the lexer was able to successfully match a string, even if it was only of length 10. This showed the lexer to work. For the lexer to be ‘useful’ however, it had to be able to match more that an input string of length 10. In this sense the lexer may have worked but it was not ‘useful’ (not usable in applications).

The optimisations increased the set of evil regular expressions the lexer could handle. As a result, the input size that could be matched increased. This lead to the lexer becoming usable in applications. When choosing the appropriate optimisations, the lexer was able to handle all the evil regular expressions tested. If however trying to generalise the lexer to handle all the evil regular expressions used in the tests, this is not the case. For each optimisation there was always a evil regular expression that the lexer couldn’t handle. The bit-code optimisation didn’t provide any benefit in terms of time or memory. It must be taken into account that the bit-code implementation in this project was only the basic optimisation. It could be thought of as a set up for the main optimisation. The simplification function specific to the bit-code optimisation was not implemented. The implementation of this simplification function, I believe, would greatly effect the run time of a bit-code optimised lexer.

Implementing the simple C lexing grammar and testing it against some of the C programs from glibc provided some of the most useful data. Although it was a short test, it showed how useful this implementation would be if it was commercialised. The derivative based lexer could be used to lex

commercial code successfully. On the other hand it showed there is still research that needs to be performed before this lexer is ready for distribution as there where many C programs that resulted in a stack overflow when lexing.

Overall the lexer was successful at lexing commercial code, and in general the optimisations greatly improved the efficiency of the lexer. In some instances, the optimised matcher was quicker and able to match larger inputs than commercial matchers. No one optimisation however provides a solution of enabling the lexer to handle all evil regular expressions.

Chapter 7

Conclusion and future work

7.1 Conclusion

The goal of the project was to take the research of Sulzmann and Lu and implement their proposed lexer [1] with different optimisations. This was successful. The lexer tokenised many different input strings using different evil regular expressions. Along with this the optimisations successfully increased the set of evil regular expressions the lexer was able to handle. There are however still many evil regular expressions and optimisations this project didn't look at that need to be investigated, as discussed in future work.

The reemergence of research into the use of derivatives is showing potential to provide a more efficient implementation for regular expressions and lexers in general. It provides many benefits over the automata implementation, for example, the ability to prove correctness. This gives good reason for continued research into the area, due to the need for faster and correct systems.

7.2 Future work

This project was a good introduction to lexers. It gave a broad view of how lexical analysis works with derivative and automata. There are many areas in this project that require continued work and research of which some are as follows:

- Bit-codes: There are still many aspects of the bitcode optimisation not implemented from the POSIX Regular Expression Parsing with Derivatives paper [13]. These being the extended regular expressions and the proposed simplification function. This would be a good starting point when continuing the work from this project.

- Simplification: As shown in the test section of this report, the derivative based Lexer [1] is not perfect. There are still evil regular expressions that cause the optimised lexer to run in exponential time. One of the main benefits of a derivative based lexer is the ease of implementing new regular expressions. This is the same case for implementing new optimisations, as seen when implementing simp2. From the work in this project, the use of simplification looked to reduce the time it took to lex strings with respect to an evil regular expressions the best. As a result, further investigation into the two simplification functions should be made. This would identify instances where they are able to optimise and where they are unable. Following on from these tests, new simplification functions should be formulated. Furthermore an investigation into how multiple simplification functions could be used together could be made to thus utilising the best optimisation for the given evil regular expression. This idea of using multiple simplification functions could also be extended to using multiple optimisations in general.

- Lexing grammars: Only a simple C lexing grammar was implemented. The investigation into the results of the C lexing grammar was confined to identifying if the implemented lexer could or could not tokenise the C program. Further investigation into the results of the C lexing rules could be made to identify why a stack overflow exception occurred. Following on from this, improvements could be made to the simple C lexing grammar. Also, lexing rules for other programming languages could be implemented to obtain a wider scope of application. Implementing the lexer in a compiler or text processor will again give informative data about how this algorithm copes when used in applications.

- Looking at commercial systems: There are already implementations of lexers using automata that are efficient enough to make regular expressions useful. The reason for this project and the research by Sulzmann and Lu was to investigate how the lexers can be optimised further. At the same time other programmers have been trying to optimise regular expressions in other ways. When providing benchmarks in the test chapter, ruby was able to match large input sizes with almost all the tested evil regular expressions. Research into how ruby implements its matching algorithms could be made. The optimisation approaches used in ruby could then be applied to the derivative base lexing algorithm. This is the same case for the newly released Java 9 matching algorithm.

References

[1] Sulzmann M, Zhuo Ming Lu K. POSIX Regular Expression Parsing with Derivatives [Internet]. Available from: http://www.home.hs-karlsruhe.de/~suma0002/publications/regex-parsing-derivatives.pdf

[2] Ausaf F, Dyckhoff R, Urban C. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl) [Internet]. Available from: https://nms.kcl.ac.uk/christian.urban/Publications/posix.pdf

[3] Brzozowski J. Derivatives of regular expressions.

[4] Urban C. Handout 1. Kings Collage London; 2018.

[5] Urban C. Compilers and Formal Languages (2). Lecture presented at; 2017; Kings Collage London.

[6] Urban C. Compilers and Formal Languages (4). Lecture presented at; 2017; Kings Collage London.

[7] OWENS S, REPPY J, TURON A. Regular-expression derivatives re-examined [Internet]. Cambridge University Press; 2009 p. 173. Available from: https://www.cs.kent.ac.uk/people/staff/sao/documents/jfp09.pdf

[8] Tobias Nipkow J. Isabelle [Internet]. Isabelle.in.tum.de. [cited 10 December 2017]. Available from: https://isabelle.in.tum.de/

[9] The GNU C Preprocessor Internals: Lexer [Internet]. Gcc.gnu.org. [cited 27 November 2017]. Available from: https://gcc.gnu.org/onlinedocs/cppinternals/Lexer.html

[10] Sulzmann M, Zhuo Ming Lu K. POSIX Regular Expression Parsing with Derivatives [Internet]. p. 2-7. Available from: http://www.home.hs-karlsruhe.de/~suma0002/publications/regex-parsing-derivatives.pdf

[11] Urban C. Compilers and Formal Languages (5). Lecture presented at; 2017; Kings Collage London.

[12] Certified Bit-Coded Regular Expression Parsing [Internet]. 2017. Available from: https://www.researchgate.net/profile/Rodrigo_Ribeiro3/publication/319162172_Certified_Bit-Coded_Regular_Expression_Parsing/links/5995d65f0f7e9b91cb095a66/Certified-Bit-Coded-Regular-Expression-Parsing.pdf

[13] Sulzmann M, Zhuo Ming Lu K. http://www.home.hs-karlsruhe.de/~suma0002/publications/regex-parsing-derivatives.pdf [Internet]. p. 11-13. Available from: http://www.home.hs-karlsruhe.de/~suma0002/publications/regex-parsing-derivatives.pdf

[14] Scala License | The Scala Programming Language [Internet]. Scala-lang.org. [cited 3 November 2017]. Available from: https://scala-lang.org/license/

[15] Programming languages — C [Internet]. 2011 p. 57-105. Available from: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[16] The GNU C Library [Internet]. Gnu.org. [cited 12 February 2018]. Available from: https://www.gnu.org/software/libc/libc.html

[17] Fernandez M. Automata and Turing Machines. Lecture presented at; 2018; Kings Collage London.

[18] Goyvaerts J. POSIX Basic and Extended Regular Expressions [Internet]. Regular-expressions.info. 2017. [cited 24 October 2017]. Available from: https://www.regular-expressions.info/posix.html

[19] Ausaf F, Dyckhoff R, Urban C. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl) [Internet]. p. 2-4. Available from: https://nms.kcl.ac.uk/christian.urban/Publications/posix.pdf

[20] Ausaf F, Dyckhoff R, Urban C. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl) [Internet]. p. 11-13. Available from: https://nms.kcl.ac.uk/christian.urban/Publications/posix.pdf

[21] C. Urban (personal communication, March 15, 2018)

[22] Kuklewicz C. Regex Posix - HaskellWiki [Internet]. Wiki.haskell.org. 2017. [cited 15 December 2017]. Available from: https://wiki.haskell.org/Regex_Posix