ABSTRACT

SHUKLA, NEERAJ CHANDRAPRAKASH. Converting Regex to Parsing Expression Grammar with Captures. (Under the direction of Dr. Jamie Jennings and Dr. Rada Chirkova).

A regular expression denotes a language accepted by some finite automata. In an effort to contrast the regular expression from automata theory with the libraries used in programming, the term 'regex', once an abbreviation, is now frequently used for a distinct class of expressions for regular and non-regular languages supported in various implementations, such as Java's *java.util.regex* or Python's *re* module.

Regexes are alleged "write-only," as they are challenging to read and understand, not only by other developers but also by their authors after some time has passed. They also vary across many "dialects", and differences between variants confuse developers. Regexes have been shown to be under-tested, with only 17% of regexes in programs being covered by test cases. Also, regexes have been the source of program bugs, including "regex denial of service". Finally, few programming language compilers do any static analysis of regexes, leading to run-time errors. Their reputation for being difficult to understand, sometimes pathological behavior, and static analysis complications suggest that they aren't suitable for large-scale projects.

Parsing Expression Grammars (PEGs), a recently introduced language recognition formal system, is based on Top-Down Parsing Language. A PEG is an alternative for regexes with more expressive power. The absence of a formal basis for regex constructions and the expressive power of PEGs led to the idea of translating regexes to PEGs. Based on PEGs, Rosie Pattern Language (RPL) is a domain-specific language for PEGs that aims to be an alternative to regex.

Prior work in converting regex to PEG renders a formalization of a simple, structure-preserving translation from explicit regular expressions to PEGs. It introduces a conversion

process for regular expressions using continuations. Regex's facilities like Captures, Back-references, Look-aheads were not considered in this prior work and posed exciting problems. Captures are retrievable sub-expression(s) in an expression. A capture is a sub-string of the original input string. This research extends the original algorithm, adding a technique for translating captures. The extended work uses the same formalization and provides the logic to generate equivalent captures in PEGs from the given regular expressions.

This thesis creates an end-to-end pipeline that starts with regex as input, converts it to a PEG, and then generates equivalent RPL expression that can match the same target strings along with retrieving the captures present in the supplied regex. The pipeline can replace regex with PEGs for text processing where production systems have a high cost of run-time errors. This implementation gives confidence that the technique for translating captures is effective in practice.

Converting Regex to Parsing Expression Grammar with Captures

by
Neeraj Chandraprakash Shukla

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina
2021

APPROVED BY:

Dr. Jamie Jennings Co-chair of Advisory Committee	Dr. Rada Chirkova Co-chair of Advisory Committee

Dr. Steffen Heber

DEDICATION

This thesis is dedicated to my father.

BIOGRAPHY

Neeraj Shukla was born on the 10th of March 1996 in Mumbai, Maharashtra, India. She received a B.Tech degree in Computer Engineering from K.J. Somaiya College of Engineering, Mumbai, in 2018. She spent a short time working with Reliance Industries Limited in Mumbai. After that, she started pursuing a master's degree in Computer Science at North Carolina State University in 2019. In her spare time, she enjoys dancing and playing games.

ACKNOWLEDGEMENTS

My words of acknowledgment can hardly begin to express my gratitude to my advisor, Dr. Jamie Jennings. I want to sincerely thank her for extending mentorship and support. Her interest in coding has inspired my work throughout. Her work in Parsing Expression Grammar, under the name 'Rosie Pattern Language,' has never ceased to amaze me. Working under her supervision has been a source of many improvements in me. I want to extend my gratitude to my committee members, Dr. Steffen Heber and Dr. Rada Chirkova, for their feedback and support. During uncertain times, the Hunt and Hill libraries, data-space, gym, and Wolfline have helped me stay motivated. I appreciate the university's management for the way they have handled COVID-19 efficiently. I am forever indebted to my parents and friends for being by my side always, and their trust has helped me grow every day.

"A few lines of recursion code, when right,
supernaturally accomplishes the whole enchilada,
where parsing that depth is human's inadequacy"

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
>
> *Jamie Zawinski*

Generative Systems, a proven asset to computer science, have made constructions possible in the areas of social network analysis, cellular automata, natural languages, etc. In theoretical computer science, a generative system of Chomsky's generative system of grammars, particularly Context-Free grammars (CFGs), is widely known. Generative Systems

define a language formally by generating strings. A recognition-based system, another well-known computer science system, is used to recognize a language in contrast to generative systems. It defines rules or predicates by which one can identify whether a given string belongs to the language or not. Most practical language applications for a recognition-based system involve recognizing, structural decomposition, or parsing of strings. A recognition-based system named Parsing Expression Grammars (PEGs) was recently founded by Ford (2004). This thesis works for advancements in Parsing Expression Grammars by expanding the already implemented work for the same. A table 1.1 is presented below to provide a piece of information, emphasizing the differences between the two systems, as mentioned earlier. Chomsky's hierarchy of grammar, shown in the image 1.1, came around 1956. From

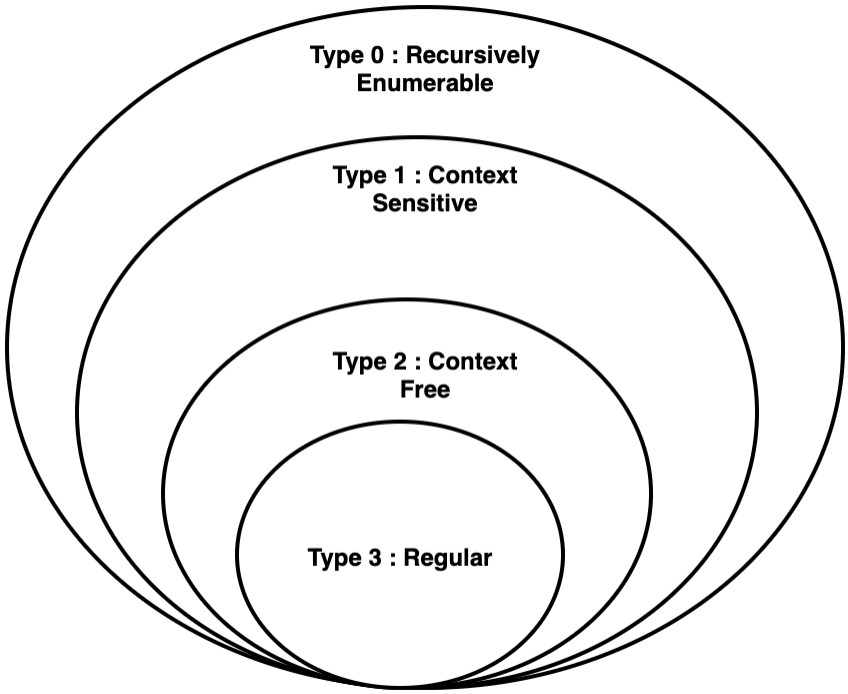

Figure 1.1: Chomsky Hierarchy

the figure, type 3 grammar, called Regular Grammar, produces regular language. Regular languages' knowledge originated the idea of regular expressions in 1951, and it is formally stated that they are capable of describing the regular languages. Entering a popular use from 1968, with one of the first appearances in the Thompson's model for pattern matching, regular expressions were later added with more functionalities with one of the results being "Global search for Regular Expression and Print matching lines" - grep (a popular search tool). Later, various other extensions were added, and every programming language started to make its own implementation of regular expressions in order to have a better fit. In an effort to contrast the regular expression from automata theory with the libraries used in programming, the term 'regex,' once an abbreviation, is now frequently used for a distinct class of expressions for regular and non-regular languages supported in various implementations, such as Java's *java.util.regex* or Python's *re* module.

Regexes are widely used to describe several patterns in practice but they are notoriously hard to read and maintain, reasons mainly being as follows:

- Dense, **cryptic** syntax
- Semantics **vary** across the implementations
- Flags that **affect** the semantics are not part of the pattern
- Regex **do not easily compose**

Parsing Expression Grammars (PEGs) (Ford 2004) are formal recognition-based systems and said to be more expressive and powerful than regex. PEGs are written syntactically, similar to Context-Free Grammars (CFGs). A linear-time parser can be developed for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing (Ford 2006). The absence of a formal basis for regex constructions and the expressive power of PEGs led to the idea of translating regexes to PEGs. Sergio

Medeiros's thesis (Medeiros et al. 2012) presented a new algorithm, Continuation Based Conversion (CBC) algorithm, to convert the regular expressions into an equivalent PEG. Certain regex's capabilities that are gained mainly by added extensions, such as look-ahead, back-references, and captures, pose an interesting threat to the CBC algorithm.

This research accounts for extending the original CBC algorithm. Regexes with captures can be converted to an equivalent PEG (if we enhance the PEG definition to include captures). One more result of the translation process is better-expressed semantics for PEGs. The idea of using a PEG, which can have a universal model of working for all the programming languages, for executing regexes led to the necessity of the development of a pipeline that can convert a regex to a PEG. If we can automatically translate a regex into an equivalent PEG, we can take advantage of tools made for PEG analysis and PEG matching. Rosie Pattern Language (RPL), a pattern generating language, is based on PEGs. It is introduced in this thesis and is shown how the PEGs can produce expressions using RPL that look like regexes.

1.1 Research Questions & Goal

Below are the research questions that this thesis is addressing.

1. Are PEGs sufficient to replace regex? If yes, can we convert all types of regex to their equivalent PEG?

2. Can the CBC algorithm, given by (Medeiros et al. 2012) be enhanced, include conversion of captures present in regex?

3. Can we take the PEGs that are resulting from the conversion of regexes with captures and use them in existing PEG matching system, e.g. Rosie Project?

The work's goal is to achieve answers to the above-mentioned research questions, present them to the readers, and accordingly implement and test a required system.

1.2 Thesis Organization

The chapter 2 of this thesis talks about the motivation behind converting regexes by introducing the regex in the ontology section and talking about the way they are used. Then in the last part of chapter, a summary of the motivation is provided. The chapter 3 of the thesis renders all the background familiarity a reader might need before advancing to the thesis. This includes two major topics, PEGs and RPL. The chapter 4 of this thesis is the literature review of the prior work. In this chapter, the work done by Ford (2004) and Medeiros et al. (2012) is mentioned in detail with an analysis. The chapter 5 mentions the CBC algorithm's enhancement, working of the same on some examples, new semantics for PEG, proof of correctness of the added part. The chapter 6 shows the implementation pipeline for converting regex to PEG with all the components' results and analysis. The chapter 7 concludes the thesis by talking about applications of the work in real life, extensibility of the pipeline, and future work.

Table 1.1: Introducing systems along with their differences

Systems	
Generative System	Recognition-based System
• Defines rules to be applied recursively to generate strings of the language.	• Defines rules or predicates that decide whether or not a given string is in the language.
• E.g., $S \longrightarrow aaS \mid \varepsilon$ is a generative definition of a trivial language over an unary character set, whose strings are "constructed" by concatenating pairs of a's.	• E.g., A regular expression $(aa)^*$ is a recognition-based definition of the same language, in which a string of a's is "accepted" if its length is even.
• Can be called as prescriptive grammar, where grammar causes language.	• Can be called as descriptive grammar, where language causes grammar.
• Good for expressing Natural Languages.	• Good for expressing Machine-Oriented Languages.
• CFG is a generative system, which for the same example mentioned above, looks like $S \longrightarrow aaS \mid \varepsilon$.	• PEG is a recognition-based system, which for the same example mentioned above, looks like $S \longleftarrow aaS \,/\, \varepsilon$.

CHAPTER

2

MOTIVATION

2.1 Ontology

A regular expression is a powerful and time-saving mechanism for solving pattern-matching problems. It is written based on mathematical theory and can describe some portion of text. In the theory of computer science, regular expressions are a compact textual representation of a regular language. The operators in a regular expression can be concatenation, alternation (choice), and repetition (denotes the repetition of an expression 0 or more times). Table 2.1 shows operators in a regular expression.

Over the last 50 years or so, as the regular expressions became ubiquitous as a text-

Table 2.1: Operators in Regular Expression

Expression	Language of Expression	Operation Performed
ϕ	ϕ	Empty Language
ε	$\{\varepsilon\}$	Empty String
a	$\{a\}$	Alphabet symbol a
$e_1 \mid e_2$	$L(e_1) U L(e_2)$	Choice/Alternations
$e_1 e_2$	$L(e_1) L(e_2)$	Concatenation
e^*	$L(e^*)$	Repetition

matching tool, their implementations grew significantly in size and complexity. Special cases, such as when an expression is a fixed string of literal characters containing no operators, are now routinely handled using an entirely different algorithm. Convenience features, such as character set expressions like [a-z] (serving as shorthand for {a, b, c, ..., z}), were early additions to the regular expression syntax, plus did not alter its expressivity or power. In other words, such additions do not enable regular expressions to match any non-regular language. Other additions to regular expression implementations were orthogonal to expressivity. The ability to capture the portion of the text matched by sub-expressions has proven immensely useful. It is supported nearly universally in recent decades. Ultimately, many additions to regular expression implementations did, in fact, increase their power. Many implementations currently support back-references, look-around, subroutines, and recursion. Each on its own brings the power to recognize some non-regular languages. Such features have received little formal analysis, particularly in combination with each other. The additional expressivity gained is not well characterized.

To contrast the regular expression that denotes a regular language from the kinds of expressions in popular use today, some now use the term **regex**, once an abbreviation, for the distinct class of expressions supported in a variety of implementations, using capabilities like back-references and recursion to express regular and many non-regular languages.

The term "regexes" is used to pronounce the plural form of the regex. We will continue this emerging use of regex and reserve the term regular expression for the denotation of a regular language.

2.2 Regex as it is used

Regex is found within the text of programs in many languages, where they are employed to validate input and extract information from a larger piece of text. Input validation typically consists of ensuring that a string's entirety matches a pattern expressed by a regex. Here, the regex denotes the language of all acceptable inputs. Sometimes validation means-testing for illegal inputs, which are then rejected. The regex, in this case, represents the language of unacceptable inputs.

Many other use cases devolve simply to extracting the parts of a text that match a regex or a sub-expression of a regex. These use cases closely resemble how regex is used on the shell command line, i.e., to search for text matching a particular pattern. Undoubtedly owing to Thompson's 1968 implementation of text search using regular expressions, when slow teletype terminals provided access to centralized time-sharing computers, regex is concise to the degree that many people find them cryptic.

Prior work has demonstrated emphatically that developers make many mistakes when writing regex (IV et al. 2019). In the software development culture, regex is sometimes described as "write-only" because they are difficult to read and understand – both by other developers and by their author after some time has passed. Consequently, it is no surprise that regex is the source of program bugs (Spishak et al. 2012). We also know that regex is under-tested (Wang and Stolee 2018), which likely contributes to regex bugs in production systems rather than during development. Moreover, many regex implementations are subject to "pathological backtracking", in which the wrong combination of regex and input

text triggers exponential-time computation during matching. Pathological backtracking is covered in detail ahead. When a user can supply the regex, the pathological behavior can be triggered in a "regex denial of service" attack. Such attacks aim to consume resources on the application server to prevent useful work from occurring Davis et al. (2018).

Despite their security issues, there is evidence that regex is very widely used. For example, Chapman and Stolee (2016) found a median of 5 (an average of 32) non-duplicate regex in randomly selected Python projects on GitHub. The popularity of question-and-answer sites like [1]Stack Overflow has provided developers with a way to get help when writing or decipher regex. Answers posted and on other sites frequently find their way into open-source programs, as illustrated in Davis et al. (2019a). However, the same work Davis et al. (2019a) clarifies what many online resources like [2]online regex tool have been shouting for years: each programming language supports a unique regex dialect, and some support more than one. For example, a regex written in a Perl program may or may not be valid in a Python or Java program. Moreover, if it is valid, it may not match the same set of strings. Furthermore, if it does, it may behave differently in terms of performance. It can also be given different results for capturing groups, like nested captures might be skipped in some implementations or might provide just the most outer captured value. The details of what captures are in the chapter 5.

From IV et al. (2019), we know that developers are frequently unaware of portability and security risks when the regex is reused. Nevertheless, Davis et al. (2019a) provides evidence, through both developer surveys and empirical analyses of open-source projects, that regex is, in fact, reused across languages.

[1]https://stackoverflow.com
[2]https://regex101.com/

2.3 Why convert Regex?

Studies of regex usage tend to focus on issues within the scope of a single project, looking at how regex is used and tested. Longitudinal studies such as Davis et al. (2019b) report that regex is involved in bugs and enhancements, also within a project but looking across time. Clearly, looking at regex portability (Davis et al. 2019a) is a significant step towards examining regex use in the large. Programming in the large refers to software development over time, involving many people, and suggests a large base of code as well. Projects in this category include enterprise software developed for internal use, software sold for local installation, software offered as a service, and of course, a wide range of consumer applications from high-end games to tax preparation tools. Some technical aspects of usages of regexes that are problematic are mentioned below.

Code Literals

Many regexes appear as code literals. If they were numbers, we would derisively label them "magic numbers" and require constant definitions for each, placed in a shared header file or module file. That is, unless and until regex are used as named constants defined in a shareable way, they will be shared by copying instead of by reference, and they will be under-tested (IV et al. 2019). In large projects, the lack of testing is exacerbated by a high-dimensional test matrix (e.g., many browser versions, many human languages, many operating systems, many server platforms). When a line of code containing a regex is hard to test at all, how likely is it to be tested across a wide swath of the test matrix?

Strings Representations

Most languages provide regex access through a library whose API accepts regex represented as strings. Compilers will, by default, ignore string literals, thus missing an opportunity to do any static analysis on the regex. The result is that some regex

bugs only surface at run-time, even ones that could have been detected statically at compile time. In projects with many developers working over a long time, there are likely to be bugs and/or new requirements that required changes to some regex. The “write only” (i.e. cryptic) nature of regex combined with under testing increases the likelihood of regex bugs. Failing to catch them early can lead to major issues in production, including long service outages (Lobo 2019).

Constructed Dynamically

Sometimes, regex are constructed dynamically, using string operations or specialized regex construction operators. In addition to thwarting easy static analysis, this type of regex usage can expose a weakness in the regex language: regex are less composable than we would like. For example, when captures are referenced by number, introducing or removing a capture invalidates numeric capture references that may be in distant parts of the code. Even when captures are named, which is not universally supported, name collisions are possible, and these may or may not be considered errors, depending on the regex implementation. If a collision is an error, the error will not be observed until run-time; if not, then the wrong data may be presented to whatever piece of code accesses the capture by the name. Generally, using specialized regex construction operators (equivalently, a regex construction API) can mitigate common errors in regex construction using string operations. They do not necessarily help with semantic errors such as changes or collisions to capture indexes.

Exponential Time - Pathological Behavior

Evaluation of a regex match is mostly carried out by simulating behaviour of equivalent Deterministic Finite Automaton (DFA) or Non-Deterministic Finite Automata (NFA) by regex engines. Most regex implementations are based on a backtracking algorithm which can exhibit pathological behavior in the form of exponential time

taken to match particular regex against particular input texts. As we noted earlier, when users enter a regex as input to a system, the system is vulnerable to a Regex Denial of Service attack. However, even when developers create the only regex used in a system, exponential behavior can still be triggered due to a general lack of awareness of the problem (Davis et al. 2019a). In the large, there may be many regex in a system that is enhanced and maintained by people over a period of time, increasing the chances that a “bad” regex will enter the system at some point.

To sum up today’s situation: regexes are widely used, under-tested, bug-prone with cryptic syntax, vulnerable to ReDOS attacks, and hard to analyze and maintain, too compact sometimes to shoehorn new features.

2.4 Moving Beyond the Current Quagmire

We may have ended up in this situation because regex today are the result of decade after decade of feature accretion. Alan Kay describes some programming languages as “agglutinations” of features, a description that can be applied to modern regex languages. The way forward is, at the risk of creating a "15th standard" [3], to look at the problem of textual pattern matching from a **Language Design** perspective. If we designed a pattern matching language to replace the one (regex) that has evolved by fits and starts away from any theoretical foundation, what would it look like? In the context of Rosie project, Dr. Jennings suggests this list of properties:

1. The syntax need not be especially compact, because modern shells, editors, and IDEs provide capable auto-completion. Also, we recognize that for programming in the large, patterns are written once but maintained over decades. If a few extra

[3] https://xkcd.com/927/

keystrokes at creation time can produce a more readable (therefore maintainable) pattern expression, it is worth the cost.

2. We develop in a polyglot world, which suggests an external (language independent) DSL for text patterns or a workable standard (or both). A critical flaw in specifications like POSIX [4] regex, is that they do not sufficiently constrain implementations that extend the standard. Also, like many programming language specifications, they say little to nothing about how features must be implemented. Just as the Scheme language standard [5] requires tail call optimization so that programmers can use recursion instead of iteration without penalty, a pattern expression specification could stipulate conditions under which linear time matching must be guaranteed.

3. Patterns must be composable and shareable. Namespaces, if not some notion of first-class pattern modules, seem necessary. First, we must allow large patterns to be built from smaller ones. In order to "hide" the building blocks that make up a complex pattern, a notion of scope would be useful. And to prevent name collisions between developers, a kind of hierarchical namespace should be supported, so that there can be a pattern named int in a package of number-recognizing patterns (e.g. num.int) and a distinct pattern named int in a package of patterns that match Unix signal names (e.g. sig.int).

   Packages, or modules, also promote pattern sharing by aggregating a set of related patterns and providing a "unit of maintenance", recognizing that developers maintain (support) applications, libraries, packages – not individual expressions or isolated lines of code.

4. Tooling that supports pattern use should fit into the modern software development

[4] https://www.regular-expressions.info/posix.html
[5] http://www.r6rs.org/

workflow. It must be possible to define patterns outside of the primary source code. Since javac neither recognizes nor analyzes regex, nor provides testing capabilities for them, we should not force developers to embed patterns in Java code, for example. Instead, named constants in source code can reference externally defined patterns within packages. Tools analogous to compilers, linters, unit test systems, and package managers can be tailored to pattern development independent of whether the primary source code language is Java or C or Go.

5. A modern DSL for pattern matching should be small relative to current regex languages in order to reduce cognitive load. The definition of any modern regex dialect is long and contains a great many exceptions to rules. E.g., PCRE[6] contains more than 9000 words for explaining the exceptions. Software developers must know the rules and the exceptions, and how they vary from one dialect to another. A complicating factor is the convention of using flags to alter regex matching behavior, which in fact alters the semantics of the regex itself.

   To properly understand a regex, then, requires intricate knowledge of the rules, the exceptions, and the circumstances of use (i.e. the flags). Instead, it should be possible to discern the semantics of an expression by looking only at the expression, i.e. with static analysis. And the pattern expression language should be kept small, ideally built out of a small set of orthogonal concepts.

2.5 Summary

Regexes are cryptic, under-tested, often used, and create many bugs; what can we do instead of using regexes? The idea of not using regexes because of their above-mentioned drawbacks

[6] https://www.pcre.org/original/doc/html/pcrepattern.html#SEC9

has motivated their conversion. A prior work has been done in Computer Science that proposes converting regular expressions into a better alternative, PEGs. This is explained in detail in the upcoming chapters. This proposal and the work have further motivated the thesis to take the task of converting regex ahead.

CHAPTER

3

BACKGROUND

Language is a process of free creation
though its laws and principles are fixed.

Noam Chomsky

3.1 Parsing Expression Grammar (PEG)

A grammar is a set of rules and can produce strings successively by re-substituting symbols. The set of strings generated from a grammar in such a way is called that grammar's language.

Putting mathematically, a language generated by a grammar **G** is a subset formally defined by

$$L(G) = \{w | w \in \Sigma^*,\ S \Rightarrow^* w\} \tag{3.1}$$

and if $L(G_1) = L(G_2)$, the Grammar G_1 is equivalent to the Grammar G_2.

Equation 3.1 has a right arrow of the grammar G for which the start symbol is S and w is any sequence of terminals from the alphabet set denoted by Σ. For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs), to express the syntax of programming languages and protocols. Syntactically similar to CFGs but having regex-like functionality, Parsing Expression Grammars (PEGs) are introduced as a recognition-based formal foundation that can decide whether a given string is in the language. PEGs are written with Extended Backus-Naur Form (EBNF) notations and models recursive descent parser practice. In simpler terms, PEGs are designed to recognize language strings whereas other grammars like CFG are generative in nature.

3.1.1 Definition of a PEG

A Parsing Expression Grammar (PEG) (Ford 2004) is formally, a 4-tuple, written as

$$G = (N, \Sigma, P_r, e_S) \tag{3.2}$$

where, N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, P_r is a finite set of parsing rules, e_S is a starting parsing expression.
The intersection of set of terminals and non-terminals is a *null* set.

Parsing rule P_r is written in the form

$$A \longleftarrow e \tag{3.3}$$

means A is a non-terminal from the set N and e is the expression that it can recognize. The rules of parsing do not have any ambiguity, they are clear, well-defined and unique. Following table summarizes the operators used to generate these parsing expressions :

Table 3.1: Operators in PEGs

Operator	name
‘ ’	literal
" "	literal
[ ]	character class
.	any character
e_1e_2	concatenation
e_1/e_2	ordered choice
e^*	zero or more times repetition
$e?$	optional
e^+	one or more
$!e$	not predicate
(e)	grouping/capturing
$\&e$	and predicate

Single quotes and double quotes both provide a limit to the string denoting the string's start and end. Square brackets indicate a class of characters, which can carry many characters, for example, [a-z]. The dot . acts as a wildcard entry that can match any single character. The sequence expression matches a concatenation of two expressions which looks like $e1e2$, returns fail if both are not found back to back. The choice expression $e1/e2$ matches

the first expression e_1; if not found, it matches the second expression e_2. The ?, *, and + operators behave the same in regular expressions, but they are greedy and possessive. The optional expression e? unconditionally "consumes" the text matched by e if e succeeds. The repetition expressions e^* consumes zero or all the entries present for e and e^+ consume one or all the entries for e. These repetitions match as many successive repetitions as possible. The expression $!e$ matches the absence of e, i.e., fails if e is matched. The expression $\&e$ first attempts to match e, then returns to the start of the text to be matched and only keeps the knowledge of the result of the match of the expression e.
An Example of a PEG,

$$S \longleftarrow aaS/\varepsilon \tag{3.4}$$

Above grammar is looking for even number of a's in a text. The language parsed by the it would look like ε, aa or $aaaa$, etc.

3.1.2 Why PEGs instead of Regex?

Parsing Expression Grammars (PEGs) are based on Top-Down Parsing Language (Ford 2004). A PEG has *more expressive power* than regular expression. PEG can represent all LR(k) languages, unlike regular expressions. LR(k) represents left-to-right, rightmost derivation in reverse, bottom-up, deterministic parsing. These properties can produce results in a single correct parse, without backtracking, and completely unambiguous, in linear time. PEGs can even recognize some non-context-free languages (Non-CFL), unlike regular expressions. The classic example of a non-context free grammar - $a^n b^n c^n$, cannot be parsed by a regex

but can be by a PEG, which is as follows :

$$A \longleftarrow aAb/\varepsilon \tag{3.5}$$

$$B \longleftarrow bBc/\varepsilon \tag{3.6}$$

$$S \longleftarrow \&(A!b)a^*B!. \tag{3.7}$$

The above PEG has S as the start symbol and uses look-ahead to first match $a^n b^n$ before skipping over the 'a' symbols to match $b^n c^n$. This example shows the greater expressivity of the PEGs.

People are looking for higher expressive matching, and this is why regex have eclectic extensions, which later has been established ad hoc. The Chomsky hierarchy came around in 1956, with strictly made margins for regular languages. PEGs, a 2000's year formalism, is a principled way to extend the regular expression by not losing any power and recognizing some context-free languages and context sensitive languages. The figure 3.1 shows the extension added here in the original hierarchy diagram.

When it comes to choice operators for CFGs or REs, PEGs use a *prioritized choice operator*, which means its selection among the alternatives is sensitive to the order in which they are given. e.g., If a regular expression $(x|xy)z$ is given, we can know the language of this expression would be $\{xz, xyz\}$. The equivalent PEG for this expression would be (xz/xyz). When matching for text for this expression, no backtracking would be involved at all unless the PEG parser is a Packrat Parser (Ford 2006) which works using memoization. Only two options PEG can see to parse,i.e., xz and xyz.

Another feature of PEGs is that repetitions are possessive in PEGs. For repetition, PEGs match the symbols from the input as much as they can. They act *possessive* kind of matcher.

The PEG engines can give parse trees instead of lists of matches. We can see various examples of a difference between the results produced for the same patterns by a regex

engine and a PEG engine in the chapter 6.

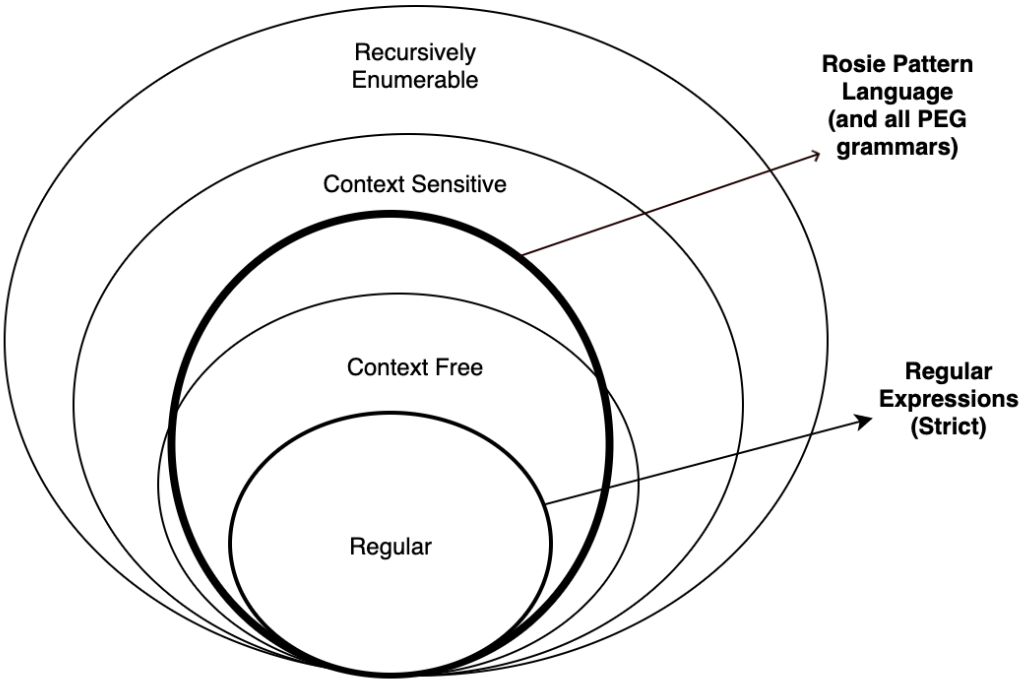

Figure 3.1: Placing PEG in Chomsky Hierarchy

3.2 Rosie Pattern Language (RPL)

For the ideology of having better readability, maintenance, and composition, Rosie was designed with a resemblance to computer programming. Rosie can develop patterns that can be stored, shared, and referenced later. These patterns are composed out of the normal, simple regex-like symbols by building smaller or unit components, and second, coupling them with a relationship like a sequence, choice, etc. It is an Expression-based language where functions are values that can be composed and revised.

How is RPL related to PEG: RPL gives a concrete syntax for PEGs, which are just mathematical constructs. It has all the capabilities of regular expressions and PEGs, along with captures and limited look behind, making it a superset of the original PEG definitions, more powerful. Rosie, like PEGs, allows recursive grammars so that RPL can recognize recursively defined structures like JSON.
The following list shows how operators of PEGs are substantiated in RPL:

- Sequences/concatenations are built by adjacency.
- Ordered Choice is expressed by /.
- Repetition operators are built using any of these *, +, ?, $\{m, n\}$ at a time.
- Look ahead is shown by >, look behind is shown by <, negation is shown by !.
- Character sets with the more strict syntax but extended composability.
- RPL has declarations, packages, macros.

Other PEG libraries that are existing are integrated with a particular programming language. In contrast, Rosie is a language-independent solution making it an elegant alternative to the ad hoc extensions to regular expressions found in most regex libraries.

CHAPTER

4

LITERATURE REVIEW

In this chapter, the thesis brings to the limelight a previous work done to transform a regular expression to it's equivalent PEG. This is mainly carrying the work done by Medeiros et al. (2012) which presents the first and original algorithm, named - **Continuation Based Conversion**, for the transformation. The algorithm is discussed in detail in the next section.

4.1 Continuation Based Conversion (CBC) Algorithm

Mederios's thesis (Medeiros et al. 2012) presented an algorithm to convert regular expressions into PEGs under the name - **Continuation Based Conversion (CBC)**. It presented multiple cases to cover a wide variety of regexes to be converted. This work's main con-

tribution is to promote regexes' execution on the formal model of PEGs. The CBC is a transformation function that accepts regex input as e, uses a continuation of that expression, and provides the same regex's parsing expression. The expression following the one to be converted is already a transformed PEG and acts as continuation. A continuation of a grammar is indeed a continuation. Continuation for all the input regex string is an empty string in the beginning. After converting a part of an expression, whatever is left is offered as the continuation and is taken ahead for the next conversion. This process continues throughout the input string, and the side-by-side PEG is built equivalent to the given regular expression. Medeiros et al. (2012) denoted this transformation function as Π, and this thesis plans to keep that same. Medeiros et al. (2012) defined four basic or atomic cases, and they are as follows:

$$\Pi(\varepsilon, k) = k \tag{4.1}$$

$$\Pi(c, k) = c\,k \tag{4.2}$$

$$\Pi(e_1 e_2, k) = \Pi(e_1, \Pi(e_2, k)) \tag{4.3}$$

$$\Pi(e_1 | e_2, k) = \Pi(e_2, k) / \Pi(e_2, k)) \tag{4.4}$$

Here, Π is a transformation function and k represents continuation of the expression.

Description of the cases are as follows :

Case 1:

This case describes 4.1. When an input expression is an empty string and has k as a continuation, the resultant PEG expression would just be the continuation directly.

Example : $\Pi(\varepsilon, a) = a$

Case 2:

This case describes 4.2. When an input expression is a character of length one and has k as a continuation, the resultant PEG expression would simply be a concatenation of regex character and k.

Example : $\Pi(a, k) = ak$

Case 3:

This case describes 4.3. When an input expression is a sequence of two sub-expressions, and has k as a continuation, then, the second sub-expression would be transformed to PEG using one of these cases and k inductively. After the the second sub-expression is transformed, the resultant PEG is used as the continuation of the first sub-expression.

Example : $\Pi(ab, c) = \Pi(a, \Pi(b, c)) = \Pi(a, bc) = abc$

Case 4:

This case describes 4.4. When an input expression is a choice of two expressions and has k as a continuation, then, the function would call itself twice, so as to convert both the first and second sub-expressions, by providing the function calls the same k as continuation. This steps results regex like $(p_1|p_2)p_3$ into parsing expression of (p_1p_3/p_2p_3)

Example : $\Pi(a|b, c) = \Pi(a, c) \,/\, \Pi(b, c) = ac/bc$

To convert a repetition e^*, the algorithm proposes the following case, which is a bit complex, hence is not atomic/basic. This conversion can take place as follows:

This is **case 5** of the algorithm.

$$e^* = e\,e^* \mid \varepsilon \tag{4.5}$$

Using the above equation, which has become more of a sequence from case 3, conversion takes place as follows:

$$\Pi(e^*, k) = \Pi(e\,e^* \mid \varepsilon,\ k) \tag{4.6}$$

$$= \Pi(e\,e^*, k) \,/\, \Pi(\varepsilon, k) \tag{4.7}$$

$$= \Pi(e\,e^*, k) \,/\, k \tag{4.8}$$

$$= \Pi(e,\ \Pi(e^*,\ k)\,) \,/\, k \tag{4.9}$$

Substituting the expression of $\Pi(e^*, k)$ for a non-terminal, let's say, A, we rewrite the 4.9 and get as follows :

$$A = \Pi(e, A) \,/\, k \tag{4.10}$$

In short, using CBC, we can convert repetition expression as follows:

$$\Pi(e^*, k) = A \tag{4.11}$$

$$where\ A \longleftarrow \Pi(e, A) \,/\, k \tag{4.12}$$

A well-formed PEG is a grammar that contains no directly or mutually left-recursive rules, such as $A \longleftarrow Aa/a$.

This is the only conversion algorithm proposed for translating regular expressions into PEGs of which we are aware. This process does not include the conversion of regex with captures present in them.

4.2 Analysis of CBC Algorithm

The algorithm of continuation based conversion reads the regex, the whole string at once first. Then, determines the type of operator and then, the case, to start with the conversion by putting k as an *empty* string. The run time complexity for the conversion is $\mathcal{O}(n)$ - linear time. This is worst case asymptotic complexity where n is the length of the supplied regex to be converted.

CHAPTER

5

EXTENDING CBC FOR CAPTURES

5.1 What are Captures?

We know that the basic and original working of any pattern matching entity is to parse the input string and provide the result of whether the input string has that pattern or not. The name capture is a functionality where a portion of the input is retrieved while pattern matching. In various languages like Python, JavaScript captures are includes in their library implementing regex. The captures are denoted using round parentheses around the expression to be captured.

5.1.1 Captures in Regex

In regex, captures are numbered, but in some implementations, captures can be named. To understand the basics first, let's see an example in python where capturing is performed in the figure 5.1. In this example, we can see that the input string of 'ac' is matched. Along

```
>>> import re
>>> r = re.compile('(a|b)c')
>>> n = r.match('ac')
>>> m = n.groups()
>>> m
('a',)
>>>
```

Figure 5.1: Captures Implementation in Python

with that, a group is generated from where we can retrieve the portion(s) of the captures string. The captures in regexes are numbered using the following two strategies.

1. **Left to Right Numbering :** For any regex expression, counting of the parentheses should start from the left and carry out until the expression's end by moving towards the right direction. Assignment of the number randomly would not provide the correct results. Examples of the numbering are as shown in the table 5.1.

2. **Depth First Numbering :** For any expression of regex, assignment of the index should be done in depth-first manner. At the 0^{th} index, the whole expression is retrieved. For nested captures, indexing should be done till we have completely numbered the outermost capturing group. Examples of the numbering are as shown in the table 5.1.

5.1.2 Captures in PEG

The formal model of PEG does not include captures. Therefore, to convert captures to PEG, we need to have a PEG concept of captures. To implement captures in PEGs, we need to write the grammar so that it can indicate the subgroup(s) to be retrieved. To write such grammar from any regex, there is a need to read the regex to check the presence of capturing groups. We can count the capturing groups in the way regex does and provide the count to the grammar. RPL is among the implementations that supports captures. Since RPL is based on recursive descent parsing in which individual parsers are named, therefore there is a natural definition of captures in RPL, which is the sub-string of the input matched by a named parser. Therefore, when we convert regex with captures, we have to annotate the PEG with each capture name.
We address numbered captures because of two reasons :

- Almost every regex implementation appears to show numbered captures. It looks like a universal practice.
- Implementing numbered captures is the hardest case, and we wanted to generalize it, as we can produce named captures from the same approach.

We have original PEGs transformations (Π) we discussed in chapter 4. This research has reformed the transformation process for covering capturing property. The new transforming expressions containing new equations and cases for captures are provided in the next section.

5.2 Transformation (Π) for Captures

The notion of converting regexes with captures into their equivalent PEGs enhances the algorithm's original transformation equations. The regexes carrying sub-expressions encap-

Table 5.1: Captures Numbering in Regex

Regex	Index	Sub-expression
(a|b)c	0	(a|b)c
	1	a|b
((xy))	0	((xy))
	1	(xy)
	2	xy
(m(xy|y)(n))	0	(m(xy|y)(n))
	1	mxy|yn
	2	xy|y
	3	n
(a(b(c)))	0	(a(b(c)))
	1	abc
	2	bc
	3	c

sulated in parenthesis showing that they have capturing groups. These capturing groups should be retrieved along during a successful parse. We can say that the parentheses denote the captures, which works basically like an operator. The expression inside the parentheses can be stated as operand of the operator of captures. Since only one operand is required for the captures, they are unary operators.

We start with a simple strategy of numbering the captures present in a regular expression. We are introducing a capture function, $\mathscr{C}$, which translates the capture information from the regex into the appropriate parts of the PEG structure. The presence of the $\mathscr{C}$ in the resulting PEG is a marker. A PEG expression marked with a capture number (or name) is intended to be understood by a PEG implementation which is capable of capturing sub-expressions of a PEG.

The additional functionality for adding captures in the conversion process has paved

the way for the following cases in the original algorithm (Jennings et al. 2019).

$$\Pi(\,(\varepsilon),\ k) = i{+}{+};\ \mathscr{C}(\Pi(\varepsilon,\varepsilon),\ i) \cdot \Pi(\varepsilon,k)$$

$$\text{Simplifies to} \tag{5.1}$$

$$\Pi(\,(\varepsilon),\ k) = i{+}{+};\ \mathscr{C}(\varepsilon,\ i) \cdot k$$

$$\Pi(\,(a),\ k) = i{+}{+};\ \mathscr{C}(\Pi(a,\varepsilon),\ i) \cdot \Pi(\varepsilon,k)$$

$$\text{Simplifies to} \tag{5.2}$$

$$\Pi(\,(a),\ k) = i{+}{+};\ \mathscr{C}(a,\ i) \cdot k$$

$$\Pi(\,(e_1 e_2),\ k) = i{+}{+};\ \mathscr{C}(\Pi(e_1,\ \Pi(e_2,\ \varepsilon)\,),\ i) \cdot \Pi(\varepsilon,k)$$

$$\text{Simplifies to} \tag{5.3}$$

$$\Pi(\,(e_1 e_2),\ k) = i{+}{+};\ \mathscr{C}(\Pi(e_1,\ \Pi(e_2,\ \varepsilon)\,),\ i) \cdot k$$

$$\Pi(\,(e_1|e_2),\ k) = i{+}{+};\ \mathscr{C}(\Pi(e_2,\ \varepsilon),\ i) \cdot \Pi(\varepsilon,k)\ /\ \mathscr{C}(\Pi(e_2,\ \varepsilon),\ i)\,.\,\Pi(\varepsilon,k)$$

$$\text{Simplifies to} \tag{5.4}$$

$$\Pi(\,(e_1|e_2),\ k) = i{+}{+};\ \mathscr{C}(\Pi(e_2,\ \varepsilon),\ i) \cdot \Pi(\varepsilon,k)\ /\ \mathscr{C}(\Pi(e_2,\ \varepsilon),\ i) \cdot k$$

$$\Pi(\,(e^*),\ k) = i{+}{+};\ \mathscr{C}(A/\varepsilon,\ \text{where } A \longleftarrow \Pi(e,A)\ /\ \Pi(e,\varepsilon),\ i) \cdot \Pi(\varepsilon,k)$$

$$\text{Simplifies to} \tag{5.5}$$

$$\Pi(\,(e^*),\ k) = i{+}{+};\ \mathscr{C}(A/\varepsilon,\ \text{where } A \longleftarrow \Pi(e,A)\ /\ \Pi(e,\varepsilon),\ i) \cdot k$$

The details of the above equations and their working are mentioned in the further sections.

Capture Function($\mathscr{C}$):

In order to add the conversion of the captures, we first enhance the formalization of the PEG itself to accommodate the procedure for designating captures in the expression. The strategy of numbering captures is the initial tryout for the same. Function $\mathscr{C}$ has the sole responsibility of saving a part/s of expression or the entire expression. If a valid pair of parentheses are present in the given regular expression, then the sub-expression encapsulated should be saved at the index. The function takes two parameters. 1. PEG transformed from e using Π function, 2. Index to store the parsing expression. The positioning of the capture function is a significant and deciding thought. The index is always initialized with a value 0. According to the universally followed convention, index 0 has to store/capture the entire expression, irrespective of the captures' explicit presence. The function's working is to create a storage space internally and collect the corresponding indexes' expression. In this way, the capture function helps to retrieve the sub-expression(s). The other enhancement of the algorithm(details deferred to the further sections) handles the index (i) value incrementing.

Below given is the description of the new cases produced during the enhancement of the algorithm. Since the first five cases are present in the chapter 4, we start from case 6 here.

Case 6 :

This case describes 5.1. When a capturing group is detected, the index i is incremented by 1. The sub-expression inside the parentheses, which is null string, is first transformed by Π with the continuation as ε, to an equivalent PEG, which is given by $\Pi(\varepsilon, \varepsilon)$. This enters the case 4.1 of the CBC. Next, the PEG is provided to the capture function and the index value where it is stored and can be later retrieved. The capture

function returns the equivalent PEG, which is juxtaposed with the continuation k of the expression.

Case 7 :

This case describes 5.2. When a capturing group is detected, the index i is incremented by 1. The sub-expression inside the parentheses - here e, is first transformed by Π with the continuation as ε, $null$, to an equivalent PEG, which is given by $\Pi(e, \varepsilon)$. This enters the case 4.2 of the CBC. Next, the PEG is provided to the capture function and the index value where it is stored and can be later retrieved. The capture function returns the equivalent PEG, which is juxtaposed with the continuation k of the expression.

Case 8:

This case describes 5.3. When a capturing group is detected, the index i is incremented by 1. The sub-expression inside the parentheses - here e_1e_2, is first transformed by Π with the continuation as ε, $null$, to an equivalent PEG, which is given by $\Pi(e_1e_2, \varepsilon)$. This enters the case 4.3 of the CBC. Next, the PEG is provided to the capture function and the index value where it is stored and can be later retrieved. The capture function returns the equivalent PEG, which is juxtaposed with the continuation k of the expression.

Case 9:

This case describes 5.4. When a capturing group is detected, the index i is incremented by 1. The sub-expression inside the parentheses - here $e_1|e_2$. Each alternative is first transformed by Π with the continuation as ε, $null$, to an equivalent PEG. Then, it is matched for the index i, and lastly, concatenated with transformation of continuation k. The same process happens to the second alternative. Here both the alternative are considered to be stored and retrieved later. We can see, the function $\mathscr{C}$ is written twice but while parsing, only once would be called. Here, we have distributed the operation

of capture rule across the "choice of sequences". We generate two PEG captures that share the same regex capture number and because PEG choice is possessive we know that in order of the entire expression to be matched, either the left or right alternatives would be matched but not both. So the PEG capture output, will have exactly one capture with the given index.

Case 10:

This case describes 5.5. Capture one or more instances of e, label this i. If there are no instance of e, captured would be an empty expression. In either of the cases, the value would be followed up by matching k. Every repetition of e would be captured at i, whereas the regex capture keeps only the last repetition capture. So the PEG gives us all of them so we can find the regex capture i by looking for the last capture with the number i. For example, let's take an expression $(a)^*$, which has an input string of *aaaa* to parse and the expression is captured as well. The way each repetition of a would be retrieved by PEG is shown in the RPL output in the output figure 1. We can see that each repetition of a is been retrieved by the name A. We can see A is appearing four times, one for each repetition of a.

Example:

The operations specified in the cases above suggests an algorithm. Several examples present in 5.2 display that:

```
{
    "s": 1,
       "subs":
         [{
             "s": 1,
               "type": "A",
               "data": "a",
               "e": 2
          },
          {
             "s": 2,
               "type": "A",
               "data": "a",
               "e": 3
          },
          {
             "s": 3,
               "type": "A",
               "data": "a",
               "e": 4
          },
          {
             "s": 4,
               "type": "A",
               "data": "a",
               "e": 5
          }],
    "type": "*",
    "data": "aaaa",
    "e": 5
}
```


Listing 1: Match Output for the input *aaaa*

Atomic : Let's take regex as $(a)b$, and initialize i with 0,

$$\Pi((a)b,\varepsilon) = \Pi(\,(a)\,,\Pi(b,\varepsilon)\,)\cdot\Pi(\varepsilon,\varepsilon) \tag{5.6}$$

$$= \Pi(\,(a),b)\cdot\varepsilon \;\ldots\ldots\ldots\ldots (from\ 4.1) \tag{5.7}$$

$$= i{+}{+};\ \mathscr{C}(\Pi(a,\varepsilon),\ 1)\cdot\Pi(\varepsilon,b) \tag{5.8}$$

$$= \mathscr{C}(a,1)\cdot b \tag{5.9}$$

$$= a\,b,\ C_1 = a \tag{5.10}$$

Sequence : Let's take regex as (ab), and initialize i with 0,

$$\Pi((ab),\varepsilon) = i{+}{+};\ \mathscr{C}(\Pi(ab\ ,\varepsilon),1)\cdot\Pi(\varepsilon,\varepsilon) \tag{5.11}$$

$$= \mathscr{C}(\Pi(a,\Pi(b,\varepsilon),1)\cdot\varepsilon \tag{5.12}$$

$$= \mathscr{C}(\Pi(a,b),1) \tag{5.13}$$

$$= \mathscr{C}(ab,1) \tag{5.14}$$

$$= ab,\ C_1 = ab \tag{5.15}$$

Choice : Let's take regex as $(a|b)$, and initialize i with 0,

$$\Pi((a|b),\varepsilon) = i{+}{+};\ \mathscr{C}(\Pi((a|b)\,,\varepsilon),\ 1)\cdot\Pi(\varepsilon,\varepsilon) \tag{5.16}$$

$$= \mathscr{C}(\Pi(a,\varepsilon),\ 1)\cdot\Pi(\varepsilon,\varepsilon)\ /\ \mathscr{C}(\Pi(b,\varepsilon),\ 1)\cdot\Pi(\varepsilon,\varepsilon) \tag{5.17}$$

$$= \mathscr{C}(a,\ 1)\cdot\varepsilon\ /\ \mathscr{C}(b,\ 1)\cdot\varepsilon \tag{5.18}$$

$$= a/b,\ C_1 = a\ or\ C_1 = b \tag{5.19}$$

Repetition : Let's take regex as (a^*), and initialize i with 0,

$$\Pi((a^*), \varepsilon) = i{+}{+};\ \mathscr{C}(A \,/\, \varepsilon, i) \cdot k \tag{5.20}$$

$$= \mathscr{C}(A \,/\, \varepsilon,\ 1) \cdot \varepsilon \tag{5.21}$$

$$= \mathscr{C}(A \,/\, \varepsilon,\ 1) \tag{5.22}$$

$$where\, A \longleftarrow \Pi(a, A) \,/\, \Pi(a, \varepsilon) \tag{5.23}$$

5.3 PEG Relation with Captures

The study is performed on the production of the grammars from the conversion function Π. The following relations are developed to illustrate the associations and provide semantics to read the grammar's functioning. These relations are based on afresh formulated PEGs where captures are also a component. New semantics always has a counter, i, which can be any other variable, starts with 0 always. Once initialized with 0, it increments encountering a valid pair of parenthesis and provides an index or marker corresponding to the capturing group. The new semantics, which is shown below, can be read as follows :

- The left side of the arrow states two things, 1. the grammar G of an expression e, 2. input string to be parsed.
- The right arrow has two components, 1. PEG, which is written on the top of the arrow, 2. counter's initialization is indicated below the arrow.
- The result of the right arrow is written next to the right arrow. If everything is consumed or matched, then ε is written, else whatever is leftover from the match is written.
- The captures are indicated in the end. C_0 has all the matched text. The rest of the capturing groups are displayed according to the value of i.

The new semantics are of two types, one for expressions without captures and another for the expressions with captures. For both the cases, there is an implied rule that the successfully parsed string would be stored at index 0. This implied capture rule is shown is type one of the semantics and for the type two, other captures are generated.
Below are the expressions of type one semantics.

$$G[\varepsilon]\ x \xrightarrow[i=0]{PEG} x,\ C_0 = \varepsilon \tag{5.24}$$

The above relation says that $G[e]$ has consumed ε (null string) while parsing and threw x. At index = 0, the whole parsed expression i.e., ε is retrieved.

$$G[a]\ ay \xrightarrow[i=0]{PEG} y,\ C_0 = a \tag{5.25}$$

The above relation says that $G[a]$ has consumed a while parsing and threw y. At index = 0, the whole parsed expression i.e., a is retrieved.

$$G[a]\ \varepsilon \xrightarrow[i=0]{PEG} fail,\ C_0 = \varepsilon \tag{5.26}$$

The above relation says that $G[a]$ can only consume a while parsing, since could not find it in the beginning of the expression, it returned a fail. Nothing to retrieve here.

$$G[a]\ by \xrightarrow[i=0]{PEG, b \neq a} fail,\ C_0 = \varepsilon \tag{5.27}$$

The above relation says that $G[a]$ can only consume a while parsing, since could not find it in the beginning of the expression, it returned a fail. Nothing to retrieve here.

$$\frac{G[e_1]\ xy \xrightarrow[i=0]{PEG} y,\ G[e_2]\ y \xrightarrow[i=0]{PEG} \varepsilon}{G[e_1 e_2]\ xy \xrightarrow[i=0]{PEG} \varepsilon,\ C_0 = xy} \tag{5.28}$$

The above relation says that $G[e_1]$ has consumed x while parsing the string xy and threw y. The grammar $G[e_2]$ has consumed y, the leftover from the last expression's parsing. In the end, a successful parse is achieved, as we get ε in the end. At index = 0, the whole parsed expression i.e., xy is retrieved.

$$\frac{G[e_1]\ x \xrightarrow[i=0]{PEG} fail,\ C_0 = \varepsilon}{G[e_1 e_2]\ x \xrightarrow[i=0]{PEG} fail,\ C_0 = \varepsilon} \tag{5.29}$$

The above relation says that $G[e_1]$ has not consumed x while parsing the string x and threw a fail. If the same grammar, $G[e_1]$, is concatenated with another grammar $G[e_2]$ will return a fail again for the string x. Nothing to retrieve here.

$$\frac{G[e_1]\ xy \xrightarrow[i=0]{PEG} y,\ C_0 = x}{G[e_1 | e_2]\ xy \xrightarrow[i=0]{PEG} y,\ C_0 = x} \tag{5.30}$$

The above relation says that $G[e_1]$ has consumed x while parsing the string xy and threw y. An ordered choice expression is written with the same grammar, $G[e_1]$, and grammar $G[e_2]$. Since $G[e_1]$ was capable of consuming x, the ordered choice can also consume x. A

leftover y is produced. At index = 0, the whole parsed expression i.e., x is retrieved.

$$\frac{G[e_1]\ x \xrightarrow[i=0]{PEG} fail,\ C_0 = \varepsilon \qquad G[e_2]\ x \xrightarrow[i=0]{PEG} \varepsilon,\ C_0 = x}{G[e_1|e_2]\ x \xrightarrow[i=0]{PEG} \varepsilon,\ C_0 = x} \tag{5.31}$$

The above relation says that $G[e_1]$ has not consumed x while parsing the string x and threw a fail. Another grammar $G[e_2]$ can consumed x while parsing the string x and reached the end of the string. An ordered choice expression is written with the same grammar, $G[e_1]$, and grammar $G[e_2]$. Even though $G[e_1]$ was not capable of consuming x, the ordered choice can consume x as $G[e_2]$ could do. The end of the string is reached. At index = 0, the whole parsed expression i.e., x is retrieved.

From here, type two of the new semantics is shown. This section has capturing groups with the index. i starting from the value j.

$$G[(e)]\ xy \xrightarrow[i=j]{PEG} y,\ C_j = x \tag{5.32}$$

The above relation says that expression e_1 should be captured since it is present around parentheses. $G[(e_1)]$ has consumed x while parsing the string xy and y is a left over. Since the matched x was a capture as well, the counter i which was initialized with 0, has incremented and became j, where j is the number of the captures found till now. If only one capturing group is found, then the value of j would be 1. The results produced from this parse, done by the PEG G, is not just the parsing but also retrieving the captures. At index = j, the whole parsed expression i.e., x is retrieved. Since it was encapsulated in parentheses, incremented the value i and stored the text x at new value of i, and the result $C_j = x$ is also

produced.

$$\frac{G[(e_1)]\ xy \xrightarrow[i=j]{PEG} y,\ C_j = x}{G[(e_1)(e_2)]\ xy \xrightarrow[i=j]{PEG} \varepsilon,\ C_j = x,\ C_{j+1} = y} \tag{5.33}$$

The above relation says that expression e_1 should be captured since it is present around parentheses. $G[(e_1)]$ has consumed x while parsing the string xy and y is a left over. Since the matched x was a capture as well, the counter i which was initialized with 0, has incremented and became j, where j is the number of the captures found till now. If only one capturing group is found, then the value of j would be 1. The results produced from this parse, done by the PEG G, is not just the parsing but also retrieving the captures. At index $= j$, the whole parsed expression i.e., x is retrieved. Since it was encapsulated in parentheses, incremented the value i and stored the text x at new value of i, and the result $C_j = x$ is also produced. This is in the first part of the relations shown. The second part has a grammar with concatenation, e_1e_2. The input given is xy. The result ε denotes the complete string is matched and end of the string is reached with no leftovers. Since, we encountered two capturing groups, (e_1) and (e_2), the value of i incremented twice. The results of a successful match, till the end, C_j has x, C_{j+1} has y.

$$\frac{G[(e_1)]\ x \xrightarrow[i=j]{PEG} \varepsilon,\ C_j = x}{G[(e_1)|e_2)]\ xy \xrightarrow[i=j]{PEG} \varepsilon,\ C_j = x} \tag{5.34}$$

The above relation says that expression e_1 should be captured since it is present around parentheses. $G[(e_1)]$ has consumed x while parsing the string xy and nothing is a left. The second part of the expression denotes, if $G[(e_1)]$ has can match x, irrespective of whether

$G[(e_2)]$ can do or not, an ordered choice can parse the same text.

$$\frac{G[(e_1)]\ x \xrightarrow[i=0]{PEG} Fails,\ G[(e_2)]\ x \xrightarrow[i=j]{PEG} \varepsilon,\ C_j = x}{G[(e_1)|(e_2)]\ x \xrightarrow[i=j]{PEG} \varepsilon,\ C_j = x} \tag{5.35}$$

The above relation says that expression e_1 should be captured and hence is present around parentheses. $G[(e_1)]$ has failed to match x but on the other side, grammar $G[(e_2)]$ has consumed x successfully. If either of the expressions, e_1 and e_2 is capable of matching x, then we can use ordered choice operator & capture the matched expression at an index j.

$$G[(e^*)]\ x \xrightarrow[i=j]{PEG} \varepsilon,\ C_j = x \tag{5.36}$$

The above relation says that expression e has captures and is a repetition as well. It parses the the whole string, being greedy and has consumed x successfully along with saving it at C_j.

The following lemma talks about when a string can be a language of an expression.

Lemma : Given a grammar $G[(e)]$ and a string x, for any string y, $x \in L(G[e])$ if and only if $G[(e)]\ xy \xrightarrow[i=k]{PEG} y,\ C_j = x$.

Proof: By induction on the pair $(G[(e)], x)$:

The values of above pair of expressions as $(\varepsilon, \varepsilon)$, and (a, a) or any single character, can be considered as the base cases. The proof for the base cases are directly implied by the equations 5.24, 5.25, 5.26, and 5.27. In other words, $\varepsilon \in L(G[\varepsilon])$ and $a \in L(G[a])$

Concatenation (the expression $G[(e)]$ contains a sequence) : In this case, let's consider two

pairs of expressions, $(G[(e_1)], x_1)$ & $(G[(e_2)], x_2)$ and given is

$$G[(e_1 e_2)]\ x_1 x_2 y \xrightarrow[i=j]{PEG} y,\ C_j = x_1 x_2 \tag{5.37}$$

then, from the equation 5.33, we can say the individual expressions also obeys the relation which can be inductively proved. It means $x_1 x_2 \in L(G[(e_1 e_2)])$.

Ordered Choice (the expression $G[(e)]$ contains an ordered choice) : In this case, let's consider two pairs of expressions, $(G[(e_1)], x_1)$ & $(G[(e_2)], x_2)$ and given is

$$G[(e_1|e_2)]\ x_1 y \xrightarrow[i=j]{PEG} y,\ C_j = x_1 \tag{5.38}$$

then, from the equation 5.34, we can say the individual expressions also obeys the relation which can be inductively proved.

It means $x_1 \in L(G[(e_1|e_2)])$. Similarly, if given

$$G[(e_1|e_2)]\ x_2 y \xrightarrow[i=j]{PEG} y,\ C_j = x_2 \tag{5.39}$$

then, from the equation 5.34, we can say the individual expressions also obeys the relation which can be inductively proved. It means $x_2 \in L(G[(e_1|e_2)])$.

5.3.1 Proof of Correctness

There are two ways of saying that the algorithm is correct. The description for the same is present below.

- **The question:** Did the extension added to the algorithm change what is matched by the PEG? No, we did not change what is matched. We can remove captures and get back the same equations from the original algorithm. Those equations are proved by

Medeiros et al. (2012) already. One way to see that this is the case to understand that the capture function simply marks its PEG argument with a marker whose value is its integer argument ($\mathscr{C}$ does not alter its PEG argument). If, as a thought experiment, we ignored captures, the $\mathscr{C}$ function could simply return its PEG argument. It is easy to see that we would get back the Medeiros et al. (2012) equations.

- **Structural Induction** is another way of stating the correctness of the extended part of the algorithm. The equations 5.40, 5.41, 5.42, 5.43 are the bases cases for the induction are as follows :

$$\Pi(\varepsilon, k) = k \tag{5.40}$$

$$\Pi((\varepsilon), k) = i{+}{+}; \mathscr{C}(\Pi(\varepsilon), i) \cdot \Pi(\varepsilon, k) \tag{5.41}$$

The above two equations show the conversion of an empty regex. k is the continuation for both of them. The first expression 5.40 is without captures and the second 5.41 is with captures. The Capture function $\mathscr{C}$ returns the equivalent PEG. which is ε and the matched using this PEG would be saved at index 1.

$$\Pi(e, k) = e \cdot k \tag{5.42}$$

$$\Pi((e), k) = i{+}{+}; \mathscr{C}(\Pi(e, \varepsilon), i) \cdot \Pi(\varepsilon, k) \tag{5.43}$$

The above two equations show the conversion of a single character expression, and k is the continuation for both of them. The single character expression e is translated using Π. The first expression 5.42 is without captures and the second 5.43 is captured

where the equivalent PEG is returned using $\mathscr{C}$.

For proving concatenation : Let's take an expression which has a concatenation of two sub-expressions, e_1 and e_2 with a captures. The conversion of the expression according to the algorithm would be as follows :

$$\Pi((e_1 e_2),\ \varepsilon) = i{+}{+}; \mathscr{C}(\ \Pi(e_1,\ \Pi(e_2, \varepsilon\ )), 1) \cdot \Pi(\varepsilon, \varepsilon) \tag{5.44}$$

$$= \mathscr{C}(\ \Pi(e_1,\ \Pi(e_2, \varepsilon\ )), 1) \cdot \varepsilon \qquad \text{from (5.40)} \tag{5.45}$$

$$= \mathscr{C}(\ \Pi(e_1,\ e_2\ ), 1) \qquad \text{from (5.42)} \tag{5.46}$$

$$= \mathscr{C}(\ e_1 e_2, 1) \qquad \text{from (5.42)} \tag{5.47}$$

Here, using base cases we can induce the expression for concatenation.

For proving choice : Let's do the same for choice :

$$\Pi((e_1|e_2),\ \varepsilon) = i{+}{+}; \mathscr{C}(\ \Pi(e_1, \varepsilon\ ), 1) \cdot \Pi(\varepsilon, k) \ /\ \mathscr{C}(\ \Pi(e_2, \varepsilon\ ), 1) \cdot \Pi(\varepsilon, k) \tag{5.48}$$

$$= \mathscr{C}(\ e_1, 1) \cdot \varepsilon\ /\ \mathscr{C}(\ e_2, 1) \cdot \varepsilon \ldots [\text{from (5.42) \& (5.40)}] \tag{5.49}$$

$$= \mathscr{C}(\ e_1, 1)\ /\ \mathscr{C}(\ e_2, 1) \tag{5.50}$$

In this way, we can induce for choice operator too. For proving kleene star : Let's take regex as (e^*):

$$\Pi((e^*), \varepsilon) = i{+}{+};\ \mathscr{C}(A\ /\ \varepsilon, i) \cdot\ k \tag{5.51}$$

$$= \mathscr{C}(A\ /\ \varepsilon,\ 1) \cdot\ \varepsilon \tag{5.52}$$

$$= \mathscr{C}(A\ /\ \varepsilon,\ 1) \tag{5.53}$$

$$\text{where } A \longleftarrow \Pi(e, A)\ /\ \Pi(e, \varepsilon)$$

A is any non-terminal which can recognize the repetition of the expression e. Here A

is given as,

$$A \longleftarrow \Pi(e, A) \;/\; \Pi(e, \varepsilon) \tag{5.54}$$

$$A \longleftarrow eA \;/\; e \qquad \text{from (5.41)} \tag{5.55}$$

Here, using the base case 5.41, we can say for the repetition as well we can implement induction and prove the enhanced algorithm is correct.

In this way, we show a method for enhancing the CBC algorithm for capturing each of the expression structures; empty string, literals, concatenations, choices, repetitions (Kleene star).

CHAPTER

6

IMPLEMENTATION & RESULTS

6.1 Pipeline

Regex conversion to PEG is a step-wise process that should start from a regex as a string, restyling it, and producing an abstract, PEG syntax tree, and concrete the PEG tree using Rosie Pattern Language executable. A pipeline is required for this conversion; as for any software delivery process, the pipelines can automatically pick up the next step upon receiving the previous steps' results. This project provides an end-to-end pipeline that starts with preprocessing the input regex, which internally involves various steps. It ends with the equivalent RPL expression that can match the same target subjects. The pipeline

is extensible since the PEG generated can be provided further to the tools capable of analyzing/testing it and using it for text matching. The components are shown in the 6.1 and details are mentioned below it. The results of each process of the pipeline are shown in detail in the section 6.2.

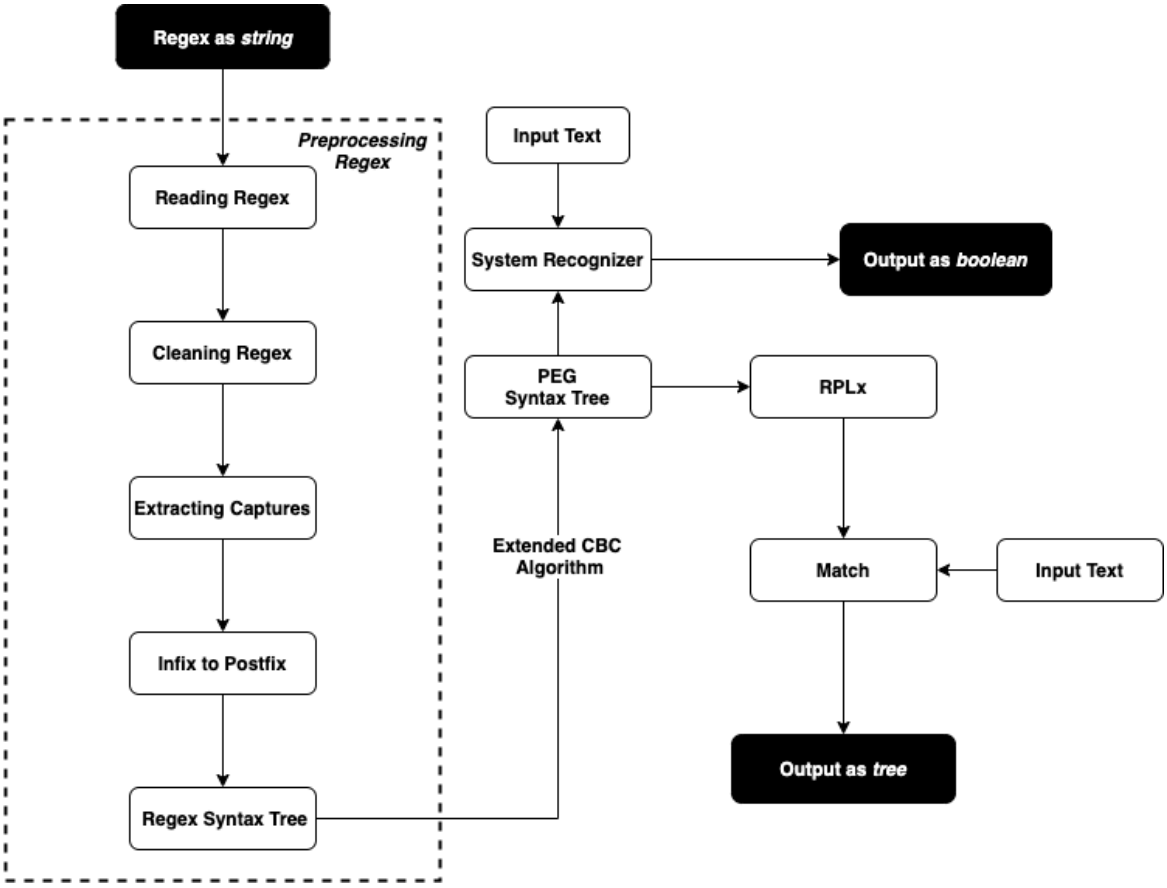

Figure 6.1: Pipeline for producing PEG and RPLx from a regex

6.1.1 Components

Preprocessing Regex

1. Reading Regex :

   Reading regex, the first step of preprocessing enables a computer program to read the supplied regex, which a whole, is a string. Different programming languages have their own set of writing regex syntax, so reading the same regexes is not a dealing step for those programs. However, when it is entering as a string in the system, reading that regex should be a defined procedure.

2. Cleaning Regex :

   Regex is cleaned before it advances in the pipeline. Cleaning is required to understand the coupling and separation of nodes from each other to deduce the direct/indirect connections between them. Eliminating meaningless/non-required nodes and adding required nodes if absent are two major activities for cleaning.

3. Extracting Captures :

   We know that regex, originally, is used for pattern matching. In practice, retrieving the portions that match the specifically asked sub-expression is also performed with captures. In this step, we are caching the group/s of expression under captures after reading the expression. Universally, in all pattern-matching implementations, parentheses are used for denoting captures. According to the research implemented by this thesis, to correctly extract the capturing groups, two ideologies are followed (discussed in chapter 5). They are Depth First Numbering and Left-to-Right numbering. This step is performed using two stacks. We perform recursion here, again, do depth-first number starting from the left of the expression. The first stack is used to keep a check on the individual symbols of the expression. The second stack is used to

store (push operation) and provide (pop operation) a correct counter for the correct expression.

4. Postfix Conversion of Regex :

   This step converts infix expression to postfix. This step makes use of the shunting-yard algorithm. It is modified a bit with operators' precedence and associativity and has an additional task on encountering capturing groups. Like basic shunting-yard, even this uses two stacks, one for the operators and another for the output. The Precedence and associativity of the operators are mentioned below. Summary of the rules used for transforming the infix expression to postfix of the regex with captures is as follows:

   (a) If the incoming symbol is a LITERAL in the expression, then push it to the output stack.

   (b) If the incoming symbol is a PIPE OPERATOR or DOT OPERATOR, then while the top of the operator stack has any operator, compare it's precedence than that of the current operator. If the precedence of the top of stack operator is greater or equal to that of the current operator, and also the current operator's associativity is RIGHT, then pop the operator from the operator stack and push it to the output stack. Push the current operator in the operator stack.

   (c) If the current symbol is a LEFT PARENTHESIS '(', then push it to both the stacks, output stack and operator stack.

   (d) If the current symbol is a RIGHT PARENTHESIS ')', then while the operator stack is not *nil* and get LEFT PARENTHESIS on the top, pop out the operator/s from the operator stack and push to the output stack sequentially. By this time, the top of the operator stack would be LEFT PARENTHESIS, pop it out and push it to the output stack.

(e) If the current symbol is a STAR OPERATOR (∗) or LAZY STAR OPERATOR (?), then push it to the output stack.

5. Regex Syntax Tree :

   An expression tree, a binary tree, has operators as internal nodes and operands as leaf nodes. Here operators like the pipe, dot, quantifying stars (lazy and greedy) are included. Each symbol from the supplied regex gets transformed to a node of the expression tree in the implementation. Each node has a set of attributes to layout its nature. Attributes are described in the table 6.1.

Generation of PEG Syntax Tree

At this step, the expression tree generated from parsing a regex is translated to a PEG syntax tree, which we can call a PEG tree using the CBC (including captures) algorithm. A node's information is updated according to the type of transformation (case of Π) applied to it. The attributes of the nodes remain the same for the PEG tree as well. The region of variation of a PEG node and regex node is the *type* attribute. The *type* attribute of a regex literal node is made as *LITERAL* whereas with PEG literal node, it is *PEG_LITERAL*. Similarly, it is happening with all the other types of nodes. The algorithm for the conversion of the Regex tree to PEG tree, Continuation Based Conversion (CBC), is been discussed in chapter 4 and chapter 5. The process of obtaining the PEG tree from the regex tree is shown along with the results in the portion 6.2.

RPL Executable (RPLx)

This step involves reading the PEG syntax tree and generating one RPL expression (RPLx) per grammar obtained from the previous step. The RPLx generated is used to them recognize the **input text** given whether it is accepted or rejected. The 6.2 shows the sequence in which

the RPL works. It starts with creating Abstract Syntax Tree (AST) for each node, the AST of the node flows to RPL compiler where a compiled pattern is created using the engine of Rosie and RPLx is produced. The match produces as result in a json format which can be seen in the section under the name results.

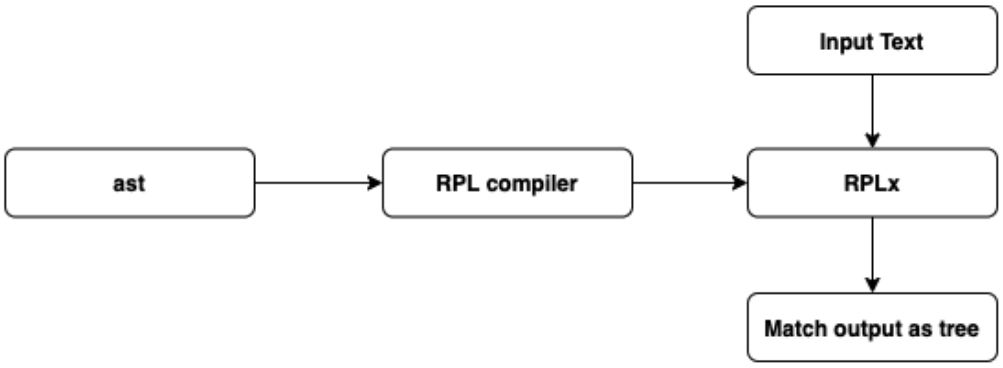

Figure 6.2: RPLx Output

System Recognizer

This part is connected to PEGs, generated as output, directly. This module can parse an input text (string) and look for the expression/s to be matched based on the PEG tree. The generated PEG tree data, especially nodes' attribute values, allow the engine to look for the target text to match. It is designed based on operators' and operands' arrangement and functions like generating expression for languages. This implementation is basically carried to get a boolean output of a match, true if the text is a language of the pattern and $false$ otherwise. Recognizer has a role in supporting testing of our complete pipeline.

6.2 Results

This section carries results of the all processes from the pipeline, and is shown component wise.

1. Preprocessing Regex:

    - Reading Regex:

      Table 6.2 shows how the code reads the supplied regex:

    - Cleaning Regex :

      The DOT (an operator for a SEQUENCE expression) operator's addition occurs between the nodes that are concatenated or juxtaposed with each other. If the nodes are already connected with any other operators like PIPE (an operator for a CHOICE expression) or brackets, the DOT is not added. This process works by reading the read regex from the previous step. Output produced is as shown in table 6.3.

    - Extracting Captures :

      The captures from the cleaned expressions are extracted at this step. The output generated by this operation is same as shown in the table 5.1.

    - Postfix Conversion :

      The postfix expressions are given in the table 6.4

    - Regex Syntax Tree :

      To show the result for how a regex syntax tree is generated, we have used one example of regex and rest of the results are also shown based on the same the example. Syntax tree for the regex - $(m(xy|y)(n))$.

      After incorporating the attributes values for each symbol in the regex, a tree node is generated for it. The regex tree generated by the code is as shown in the

image 6.3. The description of the root node in detail is given in the table 6.5. This

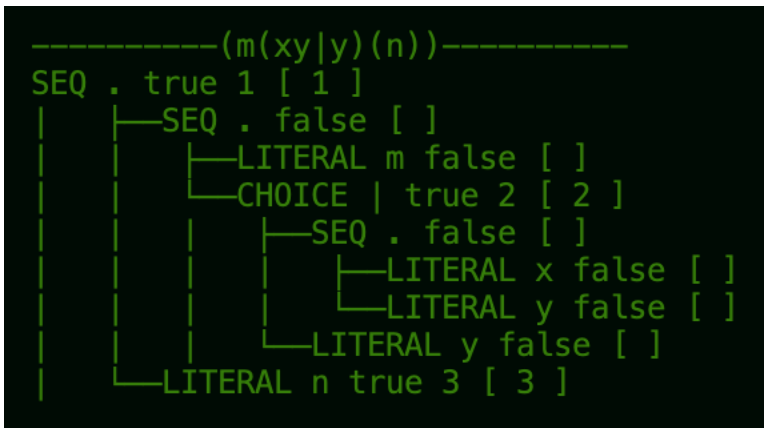

Figure 6.3: Regex Syntax Tree - The top line with dashes has the regex in a center, which is $(m(xy|y)(n))$. Each symbol of the regex is converted to a node and a tree is generated. The root of the tree here is SEQ node with data as $\cdot$. It has two children, the first from the top is a left child which is a SEQ node and the second is the right child which is a LITERAL node. Their children is shown below them. The lines made using bars (|) act as a boundary for the levels of the tree.

shows how the attributes are set and their values are read. This is same for every node.

2. PEG Syntax Tree Generation :

   The regex tree from the previous step is translated to the PEG tree, which can be drawn in a similar way as expression tree. The tree generated from the code is as shown in the image 6.4. Here we can see that, the regex is translated to a parsing expression grammar with 'S' as a start symbol.

   In the above image, we can see that, the nodes' attributes are updated, and some new nodes are added. Root node, *PEG_SEQ* node, has two children of *PEG_LITERAL* node with *data* as *m* and a *PEG_CHOICE*. *PEG_CHOICE* node is an ordered choice of two *PEG_SEQ* nodes. The capturing values of *SEQ xy* or *LITERAL y* from

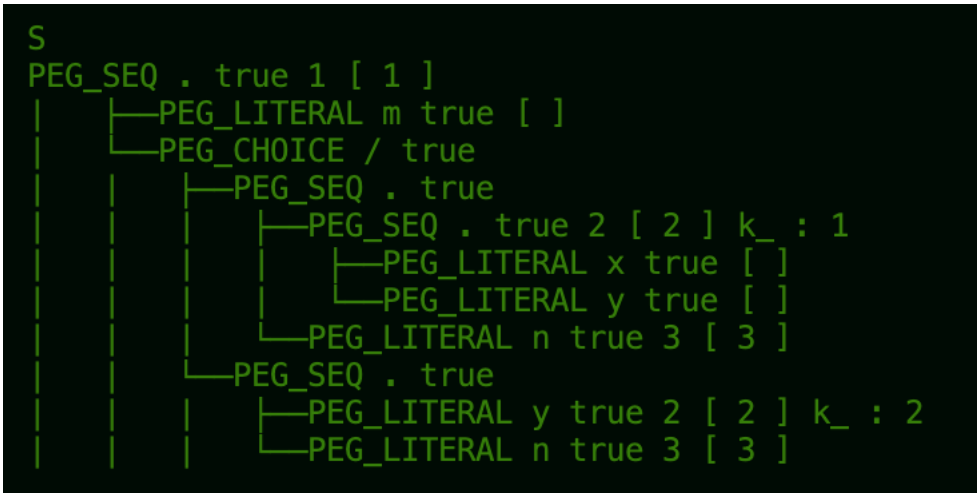

Figure 6.4: PEG Syntax Tree for the expression $(m(xy|y)(n))$. The start symbol S give the grammar present below it. The structure of the output tree is similar to that of regex tree.

the regex can be seen written as nodes with the corresponding capturing indices. First, the complete regex was under a capture, which is how the root node is having a $captureIndex$ value 1, along with $isCaptures$ boolean as $true$. Second, the choice between xy & y is captured, which is how the two nodes of PEG_SEQ (sequence of literals x and y) and $LITERAL\ y$ are carrying the value of $captureIndex$ as 2. The first part of the choice is noted with the value of $k\_$ attribute which is 1 and second part is noted as 2. That's how we can see the PEG_SEQ node with the index value 2 has a key value as 1 and the $PEG\_LITERAL$ node, index value 2 and key value as 2. Third, the Literal with $data$ value as n is marked for captures and has $captureIndex$ value as 3. Checking the last PEG_SEQ node in the image, we can see that the sequence is formed between two entities with the different capture values.

3. RPLx & Text Matching :

   RPLx provide the match results in the form of json object. The details in the json involve the properties as follows:

```
{
 e: 5,
 data: "mxyn",
 type: "C_1 ",
 subs:
   {1: {e: 4,
        data: "xy",
        type: "C_2 k_ : 1 ",
        s: 2},
    2: {e: 5,
        data: "n",
        type: "C_3 ",
        s: 4}},
 s: 1
}
```

Listing 2: Match Output for the input $mxyn$

- e : gives the end index of the matched portion
- $data$: gives the data of the matched portion
- $type$: gives the type of the matched portion
- $subs$: gives the list of the details for the other types of data present in the complete match.

Using the same regex of $(m(xy|y)(n))$, if we provide the input string of $mxyn$, the output produced is as json display 2.

Using the same regex of $(m(xy|y)(n))$, if we provide the input string of myn, the output produced is as json display 3.

A sample output of 50 regexes and their equivalent PEGs, along with an input to be matched and the results, is shown in the chapter C of appendix.

```
{
 e: 4,
 data: "myn",
 type: "C_1 ",
 subs:
    {1: {e: 3,
         data: "y",
         type: "C_2 k_ : 2 ",
         s: 2},
     2: {e: 4,
         data: "n",
         type: "C_3 ",
         s: 3}},
  s: 1
 }
```

Listing 3: Match Output for the input *myn*

6.2.1 Analysis

This section analyzes the pipeline by putting forward the time complexity of each component. The description is present in the table 6.6.

An Example of Enhancement : $a(b|c)d$

Medeiros et al. (2012)'s CBC algorithm could not convert this example into its equivalent PEG. In this implementation, we show how we do it. The image 6.5 shows the equivalent PEG. We can see that the literals b and c which are captures for the same index value i.e., 1 and hence both of them their $captureIndex$ value as 1. Since they are two different sides of the choice operator, we differentiate them with the key value, k, b is saved at 1 which a name C_1k_1 and c with a C_1k_2. The RPLx of the same is produced and the tested across two string inputs of abd and acd. The output is shown in the display 4 and 5.

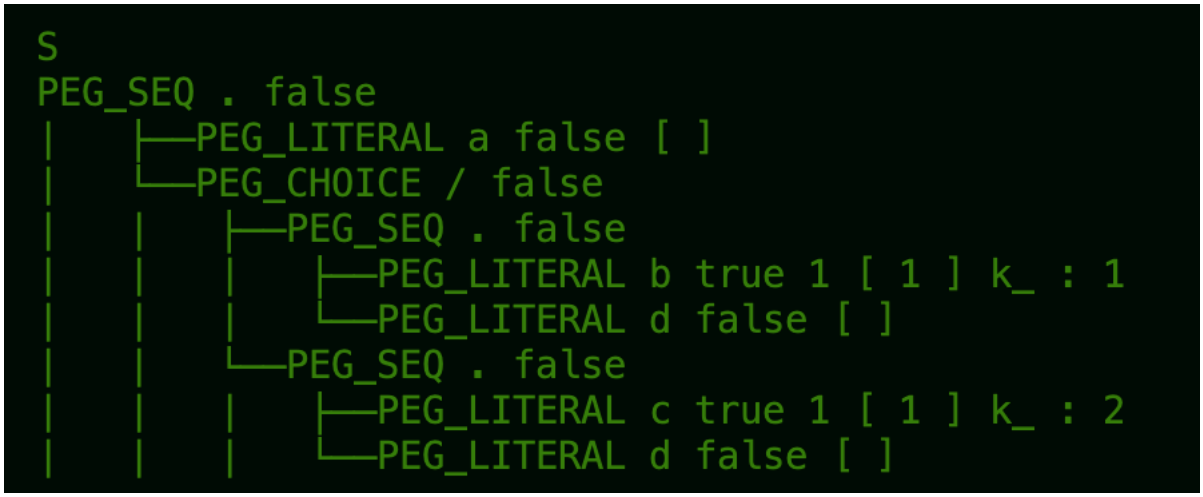

Figure 6.5: PEG form of : a(b|c)d

```
{
s: 1,
type: "*",
data: "abd",
subs:
{1: {s: 2,
    type: "C_1 k_ : 1 ",
    data: "b",
    e: 3}},
e: 4
}
```

Listing 4: Match Output for the input *abd*

```
{
 s: 1,
 type: "*",
 data: "acd",
 subs:
 {
 1: {
     s: 2,
         type: "C_1 k_ : 2 ",
         data: "c",
         e: 3
     }
 },
  e: 4
 }
```

Listing 5: Match Output for the input *acd*

Table 6.1: Node Attributes

Name	Default Value	Description	Other Values
data	0	Contains the data of the input symbol	ANY
type	*nil*	Contains the type of the input symbol	*CHOICE*, *LITERAL*, *SEQ*, *QUANTIFIER*, *STAR*, *QUESTION_MARK*, *PEG_LITERAL*, *PEG_SEQ*, *PEG_CHOICE*
left	*nil*	Is the left child of the node	*node*
right	*nil*	Is the right child of the node	*node*
isCaptures	*false*	Is a boolean value which indicates whether to capture the symbol or not	*true*
captureIndex	−1	Is an index of capture if the node is to be captured else it will be -1	Any positive integer
key	0	Is used for indicating which side of the *CHOICE* is the current node, if left then key would be 1 and if right it would be 2, 0 otherwise	1, 2
captureIndTable	*nil*	Is a table to store all the capturing index that the node belongs to	Table of positive integer values
isBinder	*false*	Indicates whether is binder or not, Binder are Non terminals	Any Capital letter alphabet
isLazy	*false*	Indicates whether the quantifier is greedy or lazy, if this is *true* it mean the quantifier is Lazy	*true*

Table 6.2: Reading Regex

Regex	Read As
a	$posix\,(\,lit\,(a)\,)$
a|b	$posix\,(\,lit\,(a)\,pipe\,lit\,(b)\,)$
ab	$posix\,(\,lit\,(a)\,lit\,(b)\,)$
(a|b)c	$posix\,(\,extended\_grammar\,(\,lit\,(a)\,pipe\,lit\,(b)\,)\,lit\,(c)\,)$
((xy))	$posix\,(\,extended\_grammar\,(\,extended\_grammar\,(\,lit\,(x)\,lit\,(y)\,)\,)\,)$
(m(xy|y)(n))	$posix\,(\,extended\_grammar\,(\,lit\,(m)\,extended\_grammar\,(\,lit\,(x)\,lit\,(y)\,pipe\,lit\,(y)\,)\,extended\_grammar\,(\,lit\,(n)\,)\,)\,)$
a*	$posix\,(\,lit\,(\,a\,star\,)\,)$
(a*?)(a*)	$posix\,(\,extended\_grammar\,(\,lit\,(a\,star\,questionmark)\,extended\_grammar\,(\,lit\,(a\,STAR)\,)\,)$

Table 6.3: Cleaning Regex

Regex	Cleaned Regex
a	a
$a\|b$	$a\|b$
ab	$a.b$
$(a\|b)c$	$(a\|b).c$
$((xy))$	$((x.y))$
$(m(xy\|y)(n))$	$(m.(x.y\|y).(n))$
$a*$	$a*$
$(a*?)(a*)$	$(a*?).(a*)$

Table 6.4: Postfix

Infix	Postfix
a	a
$a\|b$	$ab\|$
ab	$ab.$
$(a\|b)c$	$(ab\|)c.$
$((xy))$	$((xy.))$
$(m(xy\|y)(n)$	$(m(xy.y\|).(n).$
$a*$	$a*$
$(a*?)(a*)$	$(a*?)(a*).$

Table 6.5: Root Node Description

Attribute	Value
$data$	*SEQ*
$type$	*SEQ*
$left$	TreeNode
$right$	TreeNode
$isCaptures$	$true$
$captureIndex$	1
$captureIndTable$	[1]
$isBinder$	$false$
$isLazy$	$false$

Table 6.6: Analysis of the Pipeline's Components

Operation	Analysis
Reading Regex	Reads the whole string of regex for once, time complexity is $\mathcal{O}(n)$, where n is a length of the string.
Cleaning Regex	Reads the whole string of regex for once, time complexity is $\mathcal{O}(n)$, where n is a length of the string.
Extracting Captures	The time complexity is $\mathcal{O}(n)$ as it reads the cleaned regex for once and n is the length of the string. It is implemented using two stacks, space complexity is also $\mathcal{O}(n)$.
Infix to Postfix	This is infix to postfix conversion preserving the parentheses. Both the time and space complexity is $\mathcal{O}(n)$.
Regex Syntax Tree	Shunting yard is applied with few extra cases to preserve captures. Both the time and space complexity, in all the cases, is $\mathcal{O}(n)$.
PEG Syntax Tree	The CBC is applied here, which has a time complexity of a linear time, i.e, $\mathcal{O}(n)$, where n is the number of nodes in the regex tree which is to be converted to PEG tree. The function Π runs for all the nodes once and each call has a constant time complexity of $\mathcal{O}(1)$.
RPLx	RPLx generation is implemented recursively, starting from root node reaching all leaf nodes of the PEG tree. It has linear time complexity for the nodes, resulting to overall $\mathcal{O}(n)$, where n is a number of peg nodes.

CHAPTER

7

CONCLUSION & FUTURE WORK

This thesis presented work that promises to incorporate the conversion of regexes with captures into equivalent PEG. To achieve the same, the prior work done by (Medeiros et al. 2012) is investigated and provided with enhancements. These enhancements mainly focus on the captures present in the regex, which are retrieved while parsing, as operators in the expression. By treating captures as operators, we could develop a new series of steps to add to the continuation-based conversion process. Since the new PEGs can denote capturing expressions, we presented new semantics to read PEGs along with the new additions.

One way to show that the enhancements have equipped the continuation-based conversion is by showing an example. The thesis of Medeiros et al. (2012) has concluded that the continuation-based conversion is the first proposal of such a conversion but cannot

work with the expressions like **'a(b|c)d'**. It has captures around b|c, which obstructs the conversion. Here, in this thesis, we conclude that the same algorithm can now translate such regex with the following output as shown in figure 6.5 and figure 4, and is the first proposal of a conversion with captures. A study behind regexes, their functionalities, PEGs (Ford 2004), the CBC algorithm and the pipeline that we implemented concluded with the intents of expressing how, why, and where regexes are not supposed to be used in the real world and what they should be replaced so at to achieve better performance.

In short, we conclude that our implementation gives confidence that the technique for translating captures is effective in practice.

Rosie, a project based on the PEGs for pattern matching, is a language-independent library. It can be imported into any application which demands string/pattern matching. This thesis was implemented using Lua. Similarly, other programming languages can make use of Rosie, which implies that they all can make use of PEGs. The application of this research can be carried in the following way:

- Take the input regex to be converted to and provide it to the pipeline to get RPLx as an output.
- Import Rosie in the system where the parsing has to be done.
- Use the RPLx in the place where regex was placed.

This can nearly be done in all the applications systems using "write-only", low maintainable regexes. To state some examples, resume parsing for job application, looking for IP addresses, telephone number, vehicle number, etc., in a file, and many more.

Extensibility : According to our examination, the extensibility of the pipeline implemented, transforming regexes into PEGs, is quite high. It can be appended by adding PEG analysis, PEG testing, PEG matches. It can provide the resulting PEG to any other

expression-building languages like RPL to generate PEG expressions. Since the captured values in the PEG are indicated explicitly with their indices and can be retrieved easily, they can be provided to the different systems that need those capturing groups. The indices of the captures are appended with the English characters, which provision the named captures strategy.

Future Work : In the future, the conversion process can be extended to convert back-references(PEG does not include definition for back-references, so to convert we would first need to define what are back-reference in PEG), look-ahead(since peg supports look-ahead, this extension is straightforward), which regexes implement as a courtesy of various extensions by programming languages and can be called tweaks in the original regex functionalities. Implementing these new features in PEGs would make them the most powerful and expressive to date. Implementing the regex tree's conversion to the PEG tree using a machine learning algorithm can be a plan, but this might not incorporate the retrieval of captures.

By presenting the research, its implementation, application, extensibility, and work to be carried out in the future, we conclude this thesis.

REFERENCES

Chapman, C. and Stolee, K. T. (2016). Exploring regular expression usage and context in python. *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

Davis, J. C., Coghlan, C. A., Servant, F., and Lee, D. (2018). The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Davis, J. C., IV, L. G. M., Coghlan, C. A., Servant, F., and Lee, D. (2019a). Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Davis, J. C., Moyer, D., Kazerouni, A. M., and Lee, D. (2019b). Testing regex generalizability and its implications: A large-scale many-language measurement study. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In Jones, N. D. and Leroy, X., editors, *POPL*, pages 111–122. ACM.

Ford, B. (2006). Packrat parsing: Simple, powerful, lazy, linear time. *CoRR*, abs/cs/0603077.

IV, L. G. M., Donohue, J., Davis, J. C., Lee, D., and Servant, F. (2019). Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *Proceedings of the 34th International Conference on Automated Software Engineering*, pages 415–426. IEEE.

Jennings, J., Nahapetyan, A., and Salzberg, J. (2019). Continuation-based conversion of regex captures. *Unpublished manuscript*.

Lobo, S. (2019). Cloudflare rca: Major outage was a lot more than "a regular expression went bad".

Medeiros, S., Mascarenhas, F., and Ierusalimschy, R. (2012). From regexes to parsing expression grammars. *CoRR*, abs/1210.4992.

Spishak, E., Dietl, W., and Ernst, M. D. (2012). A type system for regular expressions. *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs - FTfJP 12*.

Wang, P. and Stolee, K. T. (2018). How well are regular expressions tested in the wild? *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

APPENDICES

APPENDIX

A

ACRONYMS

A summary of all acronyms is documented in Table A.1.

Table A.1: A summary of acronyms used in alphabetical order.

Acronym	Abbreviation
Continuation Based Conversion	CBC
Context Free Grammar	CFG
Deterministic Finite Automaton	DFA
Domain Specific Languages	DSL
Extended Backus-Naur Form	EBNF
Finite State Machine	FSM

Table A.1: (Continued)

Global Regular Expression Print	GREP
Non-Deterministic Finite Automaton	NFA
Parsing Expression Grammar	PEG
Rosie Pattern Language	RPL
Rosie Pattern Language Executable	RPLx

APPENDIX

B

VARIABLES

A summary of all variables is documented in Table B.1.

Table B.1: A summary of common variables and their abbreviations in alphabetical order.

Variable	Abbreviation
Grammar	G
Expression	e
Any single character	a

APPENDIX

C

SAMPLE OUTPUT

This chapter has sample output of around 50 regexes. They are converted to their equivalent PEGs. The following table C.1 has the following information :

- First column has the regex to be converted and below that has the resultant equivalent PEG.
- Third column 'Input & Match' has two things, first, the target string given as an input to the PEG to recognize and the output produced by RPLx which is shown below the input given. The output of the recognition is in a form of a tree and indication of the number of the leftover characters.

The places where the input given has not matched the PEG, means where the inputs are

invalid, match is not produced and *nil* is shown. The working and the implementation is already described in the chapter 6.

Sample Output:

Table C.1: Samples of the Conversion - Regex to PEG

Regex & PEG	Input & Match
a S PEG_LITERAL a false [ ]	a {type: "*", s: 1, data: "a", e: 2} Leftover : 0
ab S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG_LITERAL b false [ ]	ab {type: "*", s: 1, data: "ab", e: 3} Leftover : 0

Table C.1: (Continued)

abc S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL b false [ ] | | |---PEG_LITERAL c false [ ]	abc {type: "*", s: 1, data: "abc", e: 4} Leftover : 0
a|b S PEG_CHOICE / false | |---PEG_LITERAL a false [ ] | |---PEG_LITERAL b false [ ]	b {type: "*", s: 1, data: "b", e: 2} Leftover : 0

Table C.1: (Continued)

ab|cd	ab
S PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL a false [ ] | | |---PEG_LITERAL b false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL c false [ ] | | |---PEG_LITERAL d false [ ]	{type: "*", s: 1, data: "ab", e: 3} Leftover : 0
abc|xyz	abc
S PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL a false [ ] | | |---PEG_SEQ . false | | | |---PEG_LITERAL b false [ ] | | | |---PEG_LITERAL c false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL x false [ ] | | |---PEG_SEQ . false | | | |---PEG_LITERAL y false [ ] | | | |---PEG_LITERAL z false [ ]	{s: 1, data: "abc", type: "*", e: 4} Leftover : 0

Table C.1: (Continued)

(a)	a
S PEG_LITERAL a true 1 [1]	{s: 1, data: "a", type: "C_1 ", e: 2} Leftover : 0
(ab)	ab
S PEG_SEQ . true 1 [1] | |---PEG_LITERAL a true [ ] | |---PEG_LITERAL b true [ ]	{s: 1, data: "ab", type: "C_1 ", e: 3}

Table C.1: (Continued)

(a|b) S PEG_CHOICE / false | |---PEG_LITERAL a true 1 [1] k_ : 1 | |---PEG_LITERAL b true 1 [1] k_ : 2	b {s: 1, data: "b", subs: {1: {s: 1, data: "b", type: "C_1 k_ : 2 ", e: 2}}, type: "*", e: 2} Leftover : 0
a(b)|c(d) S PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL a false [ ] | | |---PEG_LITERAL b true 1 [1] | |---PEG_SEQ . false | | |---PEG_LITERAL c false [ ] | | |---PEG_LITERAL d true 2 [2]	ab {s: 1, data: "ab", subs: {1: {s: 2, data: "b", type: "C_1 ", e: 3}}, type: "*", e: 3} Leftover : 0

Table C.1: (Continued)

(a|b)c	ac
S	{subs:
PEG_CHOICE / false	{1:
| |---PEG_SEQ . false	{e: 2,
| | |---PEG_LITERAL a true 1 [1] k_ : 1	s: 1,
| | |---PEG_LITERAL c false [ ]	type: "C_1 k_ : 1 ",
| |---PEG_SEQ . false	data: "a"}
| | |---PEG_LITERAL b true 1 [1] k_ : 2	},
| | |---PEG_LITERAL c false [ ]	s: 1,
	data: "ac",
	type: "*",
	e: 3}
	Leftover : 0

Table C.1: (Continued)

(a)(b)	ab
S	{subs:
PEG_SEQ . false	{1: {e: 2,
| |---PEG_LITERAL a true 1 [1]	s: 1,
| |---PEG_LITERAL b true 2 [2]	type: "C_1 ",
	data: "a"},
	2: {e: 3,
	s: 2,
	type: "C_2 ",
	data: "b"}},
	s: 1,
	data: "ab",
	type: "*",
	e: 3}
	Leftover : 0

Table C.1: (Continued)

(abc)|(xyz)	pqr
S	Error: nil table
PEG_CHOICE / false	Leftover : 1
| |---PEG_SEQ . true 1 [1]	
| | |---PEG_LITERAL a true [ ]	
| | |---PEG_SEQ . true	
| | | |---PEG_LITERAL b true [ ]	
| | | |---PEG_LITERAL c true [ ]	
| |---PEG_SEQ . true 2 [2]	
| | |---PEG_LITERAL x true [ ]	
| | |---PEG_SEQ . true	
| | | |---PEG_LITERAL y true [ ]	
| | | |---PEG_LITERAL z true [ ]	

Table C.1: (Continued)

a(b)c	abc
S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL b true 1 [1] | | |---PEG_LITERAL c false [ ]	{subs: {1: {e: 3, s: 2, type: "C_1 ", data: "b"}}, s: 1, data: "abc", type: "*", e: 4} Leftover : 0
((xy))	xy
S PEG_SEQ . true 1 [2 1] | |---PEG_LITERAL x true [ ] | |---PEG_LITERAL y true [ ]	{subs: {1: {e: 3, s: 1, type: "C_2 ", data: "xy"}}, s: 1, data: "xy", type: "C_1", e: 3}

Table C.1: (Continued)

p(q|r)	pq
S PEG_SEQ . false | |---PEG_LITERAL p false [ ] | |---PEG_CHOICE / false | | |---PEG_LITERAL q true 1 [1] k_ : 1 | | |---PEG_LITERAL r true 1 [1] k_ : 2	{type: "*", data: "pq", e: 3, s: 1, subs: {1: {type: "C_1 k_ : 1 ", data: "q", e: 3, s: 2}}} Leftover : 0

Table C.1: (Continued)

ab(c|d)	abc
S	{type: "*",
PEG_SEQ . false	data: "abc",
| |---PEG_LITERAL a false [ ]	e: 4,
| |---PEG_SEQ . false	s: 1,
| | |---PEG_LITERAL b false [ ]	subs:
| | |---PEG_CHOICE / false	{1:
| | | |---PEG_LITERAL c true 1 [1] k_ : 1	{type: "C_1 k_ : 1 ",
| | | |---PEG_LITERAL d true 1 [1] k_ : 2	data: "c",
	e: 4,
	s: 3}}}
	Leftover : 0

Table C.1: (Continued)

a(b)(c)d	abcd
S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL b true 1 [1] | | |---PEG_SEQ . false | | | |---PEG_LITERAL c true 2 [2] | | | |---PEG_LITERAL d false [ ]	{type: "*", data: "abcd", e: 5, s: 1, subs: {1: {type: "C_1 ", data: "b", e: 3, s: 2}, 2: {type: "C_2 ", data: "c", e: 4, s: 3}}} Leftover : 0

Table C.1: (Continued)

(a|b)(c|d)	ab
S	Error: nil table
PEG_CHOICE / false	Leftover : 1
| |---PEG_SEQ . false	
| | |---PEG_LITERAL a true 1 [1] k_ : 1	
| | |---PEG_CHOICE / false	
| | | |---PEG_LITERAL c true 2 [2] k_ : 1	
| | | |---PEG_LITERAL d true 2 [2] k_ : 2	
| |---PEG_SEQ . false	
| | |---PEG_LITERAL b true 1 [1] k_ : 2	
| | |---PEG_CHOICE / false	
| | | |---PEG_LITERAL c true 2 [2] k_ : 1	
| | | |---PEG_LITERAL d true 2 [2] k_ : 2	

Table C.1: (Continued)

a*	aa
A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL a false [ ] | | |---PEG A false | |---PEG_LITERAL false S PEG A false	{s: 1, data: "aa", type: "A", subs: {1: {s: 2, data: "a", type: "A", subs: {1: {s: 3, data: "", type: "A", e: 3}}, e: 3}}, e: 3}

Table C.1: (Continued)

a*|b*	b
A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL a false [ ] | | |---PEG A false | |---PEG_LITERAL false B PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL b false [ ] | | |---PEG B false | |---PEG_LITERAL false S PEG_CHOICE / false | |---PEG A false | |---PEG B false	{s: 1, data: "", type: "*", subs: {1: {s: 1, data: "", type: "B", e: 1}}, e: 1} Leftover : 0

Table C.1: (Continued)

(ab)*|(cd)*	ab
A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_SEQ . true 1 [1] | | | |---PEG_LITERAL a true [ ] | | | |---PEG_LITERAL b true [ ] | | |---PEG A false | |---PEG_LITERAL false B PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_SEQ . true 2 [2] | | | |---PEG_LITERAL c true [ ] | | | |---PEG_LITERAL d true [ ] | | |---PEG B false | |---PEG_LITERAL false S PEG_CHOICE / false | |---PEG A false | |---PEG B false	{s: 1, data: "ab", type: "*", subs: {1: {s: 1, data: "ab", type: "A", subs: {1: {s: 1, data: "ab", type: "C_1 ", e: 3}, 2: {s: 3, data: "", type: "A", e: 3}}, e: 3}}, e: 3} Leftover : 0

Table C.1: (Continued)

(ba|a)*a	a
S PEG A false A PEG_CHOICE / false | |---PEG_CHOICE / false | | |---PEG_SEQ . false | | | |---PEG_SEQ . true 1 [1] k_ : 1 | | | | |---PEG_LITERAL b true [ ] | | | | |---PEG_LITERAL a true [ ] | | | |---PEG A false | | |---PEG_SEQ . false | | | |---PEG_LITERAL a true 1 [1] k_ : 2 | | | |---PEG A false | |---PEG_LITERAL a false [ ]	{s: 1, data: "a", type: "A", e: 2}

Table C.1: (Continued)

a*?b	aab
S PEG A false A PEG_CHOICE / false | |---PEG_LITERAL b false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL a false [ ] | | |---PEG A false	{s: 1, data: "aab", type: "A", subs: {1: {s: 2, data: "ab", type: "A", subs: {1: {s: 3, data: "b", type: "A", e: 4}}, e: 4}}, e: 4}

Table C.1: (Continued)

a*?bc	abc
S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG A false A PEG_CHOICE / false | |---PEG_SEQ . true 1 [1] k_ : 1 | | |---PEG_LITERAL b true [ ] | | |---PEG A true | |---PEG_LITERAL c true 1 [1] k_ : 2	{e: 4, s: 1, type: "A", data: "abc", subs: {1: {e: 4, s: 2, type: "A", data: "bc"}}}
a(b|c)d	acd
S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG_CHOICE / false | | |---PEG_SEQ . false | | | |---PEG_LITERAL b true 1 [1] k_ : 1 | | | |---PEG_LITERAL d false [ ] | | |---PEG_SEQ . false | | | |---PEG_LITERAL c true 1 [1] k_ : 2 | | | |---PEG_LITERAL d false [ ]	{e: 4, s: 1, type: "*", data: "acd", subs: {1: {e: 3, s: 2, type: "C_1 k_ : 2 ", data: "c"}}} Leftover : 0

Table C.1: (Continued)

a*b	aab
S PEG A false A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL a false [ ] | | |---PEG A false | |---PEG_LITERAL b false [ ]	{e: 4, s: 1, type: "A", data: "aab", subs: {1: {e: 4, s: 2, type: "A", data: "ab", subs: {1: {e: 4, s: 3, type: "A", data: "b"}}}}}

Table C.1: (Continued)

xyz*	xy
S	{type: "*",
PEG_SEQ . false	s: 1,
| |---PEG_LITERAL x false [ ]	e: 3,
| |---PEG_SEQ . false	subs:
| | |---PEG_LITERAL y false [ ]	{1: {type: "A",
| | |---PEG A false	s: 3,
A	e: 3,
PEG_CHOICE / false	data: ""}},
| |---PEG_SEQ . false	data: "xy"}
| | |---PEG_LITERAL z false [ ]	Leftover : 0
| | |---PEG A false	
| |---PEG_LITERAL false	

Table C.1: (Continued)

ab*(c|d)	acd
S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG A false A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL b false [ ] | | |---PEG A false | |---PEG_CHOICE / false | | |---PEG_LITERAL c true 1 [1] k_ : 1 | | |---PEG_LITERAL d true 1 [1] k_ : 2	{type: "*", e: 3, subs: {1: { type: "A", e: 3, subs: {1: {type: "C_1 k_ : 1 ", e: 3, data: "c", s: 2}}, data: "c", s: 2}}, data: "ac", s: 1} Leftover : 1

Table C.1: (Continued)

(a|b)*(c|d)	ab
S	Error: nil table
PEG A false	
A	
PEG_CHOICE / false	
| |---PEG_CHOICE / false	
| | |---PEG_SEQ . false	
| | | |---PEG_LITERAL a true 1 [1] k_ : 1	
| | | |---PEG A false	
| | |---PEG_SEQ . false	
| | | |---PEG_LITERAL b true 1 [1] k_ : 2	
| | | |---PEG A false	
| |---PEG_CHOICE / false	
| | |---PEG_LITERAL c true 2 [2] k_ : 1	
| | |---PEG_LITERAL d true 2 [2] k_ : 2	

Table C.1: (Continued)

a(b)(c)d*	abcd
S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL b true 1 [1] | | |---PEG_SEQ . false | | | |---PEG_LITERAL c true 2 [2] | | | |---PEG A false A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL d false [ ] | | |---PEG A false | |---PEG_LITERAL false	{type: "*", e: 5, subs: {1: {type: "C_1", e: 3, data: "b", s: 2}, 2: {type: "C_2", e: 4, data: "c", s: 3}, 3: {type: "A", e: 5, subs: {1: {type: "A", e: 5, data: "", s: 5}}, data: "d", s: 4}}, data: "abcd", s: 1} Leftover : 0

Table C.1: (Continued)

(a|b)*(c|d)	cd
S	{subs:
PEG A false	{1:
A	{data: "c",
PEG_CHOICE / false	e: 2,
| |---PEG_CHOICE / false	type: "C_2 k_ : 1 ",
| | |---PEG_SEQ . false	s: 1}},
| | | |---PEG_LITERAL a true 1 [1]	data: "c",
| | | |---PEG A false	e: 2,
| | |---PEG_SEQ . false	type: "A",
| | | |---PEG_LITERAL b true 1 [1]	s: 1}
| | | |---PEG A false	
| |---PEG_CHOICE / false	
| | |---PEG_LITERAL c true 2 [2] k_ : 1	
| | |---PEG_LITERAL d true 2 [2] k_ : 2	

Table C.1: (Continued)

ab*(c|d)	abc
S	{subs:
PEG_SEQ . false	{1: {
| |---PEG_LITERAL a false [ ]	subs:
| |---PEG A false	{1: {
A	subs:
PEG_CHOICE / false	{1:
| |---PEG_SEQ . false	{data: "c",
| | |---PEG_LITERAL b false [ ]	e: 4,
| | |---PEG A false	type: "C_1 k_ : 1 ",
| |---PEG_CHOICE / false	s: 3}},
| | |---PEG_LITERAL c true 1 [1] k_ : 1	data: "c",
| | |---PEG_LITERAL d true 1 [1] k_ : 2	e: 4,
	type: "A",
	s: 3}
	},
	data: "bc",
	e: 4,
	type: "A",
	s: 2}},
	data: "abc",
	e: 4,
	type: "*",
	s: 1}
	Leftover : 0

Table C.1: (Continued)

p*(q|r)	r
S	{subs:
PEG A false	{1:
A	{data: "r",
PEG_CHOICE / false	e: 2,
| |---PEG_SEQ . false	type: "C_1 k_ : 2 ",
| | |---PEG_LITERAL p false [ ]	s: 1
| | |---PEG A false	}
| |---PEG_CHOICE / false	},
| | |---PEG_LITERAL q true 1 [1] k_ : 1	data: "r",
| | |---PEG_LITERAL r true 1 [1] k_ : 2	e: 2,
	type: "A",
	s: 1}

Table C.1: (Continued)

a(b)(c)d*	abcd
S PEG_SEQ . false | |---PEG_LITERAL a false [ ] | |---PEG_SEQ . false | | |---PEG_LITERAL b true 1 [1] | | |---PEG_SEQ . false | | | |---PEG_LITERAL c true 2 [2] | | | |---PEG A false A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL d false [ ] | | |---PEG A false | |---PEG_LITERAL false	{subs: {1: {data: "b", e: 3, type: "C_1 ", s: 2}, 2: {data: "c", e: 4, type: "C_2 ", s: 3}, 3: {subs: {1: {data: "", e: 5, type: "A", s: 5} }, data: "d", e: 5, type: "A", s: 4}}, data: "abcd", e: 5, type: "*", s: 1} Leftover : 0

Table C.1: (Continued)

a(b*|c*)d	abd
A	{subs:
PEG_CHOICE / false	{1: {subs:
| |---PEG_SEQ . false	{1: {
| | |---PEG_LITERAL b false [ ]	data: "",
| | |---PEG A false	e: 3,
| |---PEG_EPSILON false	type: "A",
B	s: 3}
PEG_CHOICE / false	},
| |---PEG_SEQ . false	data: "b",
| | |---PEG_LITERAL c false [ ]	e: 3,
| | |---PEG B false	type: "A",
| |---PEG_EPSILON false	s: 2}},
S	data: "abd",
PEG_SEQ . false	e: 4,
| |---PEG_LITERAL a false [ ]	type: "*",
| |---PEG_CHOICE / false	s: 1}
| | |---PEG_SEQ . false	Leftover : 0
| | | |---PEG A true 1 [1] k_ : 1	
| | | |---PEG_LITERAL d false [ ]	
| | |---PEG_SEQ . false	
| | | |---PEG B true 1 [1] k_ : 2	
| | | |---PEG_LITERAL d false [ ]	

Table C.1: (Continued)

(a|cd*z)	cz
S	{subs:
PEG_CHOICE / false	{1: {
| |---PEG_LITERAL a true 1 [1] k_ : 1	subs:
| |---PEG_SEQ . true 1 [1] k_ : 2	{1:
| | |---PEG_LITERAL c true [ ]	{e: 3,
| | |---PEG A true	s: 2,
A	type: "A",
PEG_CHOICE / false	data: "z"}
| |---PEG_SEQ . false	},
| | |---PEG_LITERAL d false [ ]	e: 3,
| | |---PEG A false	s: 1,
| |---PEG_LITERAL z false [ ]	type: "C_1 k_ : 2 ",
	data: "cz"}},
	e: 3,
	s: 1,
	type: "*",
	data: "cz"}
	Leftover : 0

Table C.1: (Continued)

a(b)(c|d)	acd
S	Error: nil table
PEG_SEQ . false	Leftover : 1
| |---PEG_LITERAL a false [ ]	
| |---PEG_SEQ . false	
| | |---PEG_LITERAL b true 1 [1]	
| | |---PEG_CHOICE / false	
| | | |---PEG_LITERAL c true 2 [2] k_ : 1	
| | | |---PEG_LITERAL d true 2 [2] k_ : 2	

Table C.1: (Continued)

(a(b)(c))d	abcd
S PEG_SEQ . false | |---PEG_SEQ . true 1 [1] | | |---PEG_LITERAL a true [ ] | | |---PEG_SEQ . true | | | |---PEG_LITERAL b true 2 [2] | | | |---PEG_LITERAL c true 3 [3] | |---PEG_LITERAL d false [ ]	{subs: {1: {subs: {1: {e: 3, s: 2, type: "C_2 ", data: "b" }, 2: { e: 4, s: 3, type: "C_3 ", data: "c" } }, e: 4, s: 1, type: "C_1 ", data: "abc"}}, e: 5, s: 1, type: "*", data: "abcd"} Leftover : 0

Table C.1: (Continued)

(a|b)x*(y)	axy
A PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL x false [ ] | | |---PEG A false | |---PEG_LITERAL y true 2 [2] S PEG_CHOICE / false | |---PEG_SEQ . false | | |---PEG_LITERAL a true 1 [1] k_ : 1 | | |---PEG A false | |---PEG_SEQ . false | | |---PEG_LITERAL b true 1 [1] k_ : 2 | | |---PEG A false	{subs: {1: {e: 2, s: 1, type: "C_1 k_ : 1 ", data: "a"}, 2: {subs: {1: { subs: {1: {e: 4, s: 3, type: "C_2 ", data: "y"}}, e: 4, s: 3, type: "A", data: "y"} }, e: 4, s: 2, type: "A", data: "xy"}}, e: 4, s: 1, type: "*", data: "axy"} Leftover : 0

Table C.1: (Continued)

(a|b)c(xy)	acxy
S	{type: "*",
PEG_CHOICE / false	s: 1,
| |---PEG_SEQ . false	subs:
| | |---PEG_LITERAL a true 1 [1] k_ : 1	{1:
| | |---PEG_SEQ . false	{type: "C_1 k_ : 1 ",
| | | |---PEG_LITERAL c false [ ]	s: 1,
| | | |---PEG_SEQ . true 2 [2]	e: 2,
| | | | |---PEG_LITERAL x true [ ]	data: "a"},
| | | | |---PEG_LITERAL y true [ ]	2:
| |---PEG_SEQ . false	{type: "C_2 ",
| | |---PEG_LITERAL b true 1 [1] k_ : 2	s: 3,
| | |---PEG_SEQ . false	e: 5,
| | | |---PEG_LITERAL c false [ ]	data: "xy"}},
| | | |---PEG_SEQ . true 2 [2]	e: 5,
| | | | |---PEG_LITERAL x true [ ]	data: "acxy"}
| | | | |---PEG_LITERAL y true [ ]	Leftover : 0

Table C.1: (Continued)

(a)(b)(c)	abc
S PEG_SEQ . false | |---PEG_LITERAL a true 1 [1] | |---PEG_SEQ . false | | |---PEG_LITERAL b true 2 [2] | | |---PEG_LITERAL c true 3 [3]	{type: "*", s: 1, subs: {1: {type: "C_1 ", s: 1, e: 2, data: "a"}, 2: {type: "C_2 ", s: 2, e: 3, data: "b"}, 3: {type: "C_3 ", s: 3, e: 4, data: "c"}}, e: 4, data: "abc"} Leftover : 0

Table C.1: (Continued)

(a*b)|(c*d)	ab
A	{type: "*",
PEG_CHOICE / false	s: 1,
| |---PEG_SEQ . false	subs:
| | |---PEG_LITERAL a false [ ]	{1: {type: "A",
| | |---PEG A false	s: 1,
| |---PEG_LITERAL b false [ ]	subs:
B	{1: {type: "A",
PEG_CHOICE / false	s: 2,
| |---PEG_SEQ . false	e: 3,
| | |---PEG_LITERAL c false [ ]	data: "b"}},
| | |---PEG B false	e: 3,
| |---PEG_LITERAL d false [ ]	data: "ab"}},
S	e: 3,
PEG_CHOICE / false	data: "ab"}
| |---PEG A true 1 [1]	Leftover : 0
| |---PEG B true 2 [2]	

Table C.1: (Continued)

(ab)|(cd)*z	cdz
A	{type: "*",
PEG_CHOICE / false	s: 1,
| |---PEG_SEQ . false	subs:
| | |---PEG_SEQ . true 2 [2]	{1: {
| | | |---PEG_LITERAL c true [ ]	type: "A",
| | | |---PEG_LITERAL d true [ ]	s: 1,
| | |---PEG A false	subs:
| |---PEG_LITERAL z false [ ]	{1: {type: "C_2 ",
S	s: 1,
PEG_CHOICE / false	e: 3,
| |---PEG_SEQ . true 1 [1]	data: "cd"},
| | |---PEG_LITERAL a true [ ]	2: {type: "A",
| | |---PEG_LITERAL b true [ ]	s: 3,
| |---PEG A false	e: 4,
	data: "z"}},
	e: 4,
	data: "cdz"}},
	e: 4,
	data: "cdz"}
	Leftover : 0

Table C.1: (Continued)

((m)(xy|y)(n))	myn
S	{type: "C_1 ",
PEG_SEQ . true 1 [1]	s: 1,
| |---PEG_LITERAL m true 2 [2]	subs:
| |---PEG_CHOICE / true	{1:
| | |---PEG_SEQ . true	{type: "C_2 ",
| | | |---PEG_SEQ . true 3 [3] k_ : 1	s: 1,
| | | | |---PEG_LITERAL x true [ ]	e: 2,
| | | | |---PEG_LITERAL y true [ ]	data: "m"},
| | | |---PEG_LITERAL n true 4 [4]	2:
| | |---PEG_SEQ . true	{type: "C_3 k_ : 2 ",
| | | |---PEG_LITERAL y true 3 [3] k_ : 2	s: 2,
| | | |---PEG_LITERAL n true 4 [4]	e: 3,
	data: "y"},
	3:
	{type: "C_4 ",
	s: 3,
	e: 4,
	data: "n"}},
	e: 4,
	data: "myn"}

Table C.1: (Continued)

(a|b)cd(e)	bcde
S	{type: "*",
PEG_CHOICE / false	subs:
| |---PEG_SEQ . false	{1:
| | |---PEG_LITERAL a true 1 [1] k_ : 1	{type: "C_1 k_ : 2 ",
| | |---PEG_SEQ . false	e: 2,
| | | |---PEG_LITERAL c false [ ]	s: 1,
| | | |---PEG_SEQ . false	data: "b"},
| | | | |---PEG_LITERAL d false [ ]	2: {type: "C_2 ",
| | | | |---PEG_LITERAL e true 2 [2]	e: 5,
| |---PEG_SEQ . false	s: 4,
| | |---PEG_LITERAL b true 1 [1] k_ : 2	data: "e"}},
| | |---PEG_SEQ . false	e: 5,
| | | |---PEG_LITERAL c false [ ]	s: 1,
| | | |---PEG_SEQ . false	data: "bcde"}
| | | | |---PEG_LITERAL d false [ ]	Leftover : 0
| | | | |---PEG_LITERAL e true 2 [2]	

Table C.1: (Continued)

(abc)*|(xyz)*	abc
A	{type: "*",
PEG_CHOICE / false	subs:
| |---PEG_SEQ . false	{1:
| | |---PEG_SEQ . true 1 [1]	{type: "A",
| | | |---PEG_LITERAL a true [ ]	subs:
| | | |---PEG_SEQ . true	{1:
| | | | |---PEG_LITERAL b true [ ]	{type: "C_1 ",
| | | | |---PEG_LITERAL c true [ ]	e: 4,
| | |---PEG A false	s: 1,
| |---PEG_LITERAL false	data: "abc"
B	},
PEG_CHOICE / false	2:
| |---PEG_SEQ . false	{type: "A",
| | |---PEG_SEQ . true 2 [2]	e: 4,
| | | |---PEG_LITERAL x true [ ]	s: 4,
| | | |---PEG_SEQ . true	data: ""}},
| | | | |---PEG_LITERAL y true [ ]	e: 4,
| | | | |---PEG_LITERAL z true [ ]	s: 1,
| | |---PEG B false	data: "abc"}
| |---PEG_LITERAL false	},
S	e: 4,
PEG_CHOICE / false	s: 1,
| |---PEG A false	data: "abc"}
| |---PEG B false	Leftover : 0

Table C.1: (Continued)

(((xy)))	xy
S PEG_SEQ . true 1 [3 2 1] | |---PEG_LITERAL x true [ ] | |---PEG_LITERAL y true [ ]	{type: "C_1", subs: {1: { type: "C_2", subs: {1: {type: "C_3 ", e: 3, s: 1, data: "xy"}}, e: 3, s: 1, data: "xy"}}, e: 3, s: 1, data: "xy"}

Table C.1: (Continued)

x(y*)z	xz
A	{type: "*",
PEG_CHOICE / false	subs:
| |---PEG_SEQ . true 1 [1] k_ : 1	{1:
| | |---PEG_LITERAL y true [ ]	{type: "A",
| | |---PEG A true	subs:
| |---PEG_LITERAL z true 1 [1] k_ : 2	{1:
S	{
PEG_SEQ . false	type: "C_1 k_ : 2 ",
| |---PEG_LITERAL x false [ ]	e: 3,
| |---PEG A false	s: 2,
	data: "z"
	}
	},
	e: 3,
	s: 2,
	data: "z"}},
	e: 3,
	s: 1,
	data: "xz"}
	Leftover : 0

Table C.1: (Continued)

((a)(b))	ab
S PEG_SEQ . true 1 [1] | |---PEG_LITERAL a true 2 [2] | |---PEG_LITERAL b true 3 [3]	{type: "C_1 ", subs: {1: {type: "C_2 ", e: 2, s: 1, data: "a"}, 2: {type: "C_3 ", e: 3, s: 2, data: "b"}}, e: 3, s: 1, data: "ab"}

END

Thank you very much