Department of Mathematics and Computer Science
Software Engineering and Technology

Deriving Syntax Highlighting Grammars from Character-Level Context-Free Grammars: Algorithm Development, Analysis, and Future Directions

Master Thesis

Tar van Krieken

Supervisor: Prof. Dr. Jurgen Vinju

Eindhoven, December 15, 2023

Abstract

Formal language developers typically have to specify syntactic information in multiple formats: a context-free grammar used for parsing, and syntax highlighting grammars (lexing grammars) to help users work with their language. This spreads syntactic information out over multiple files with a lot of redundancy, higher maintenance costs, and risk of desynchronization. In this thesis we explore a possible solution: augmenting context-free grammars with tokenization data, and automatically deriving lexing grammars in industry formats. We look into the feasibility of such an approach by developing a grammar transformation pipeline, which transforms a specification grammar into a grammar suitable for syntax highlighting. Supporting this approach, we establish a formalization of regular-expressions that includes capture groups and lookarounds, together with important operations on these expressions. Finally, the limitations of both the pipeline and the regular expressions formalism are discussed, while also touching on possible directions for enhancement.

Contents

1 Introduction

1.1 Motivation

Formal languages are used extensively for interaction between computers and humans. Such languages have well defined syntax, as well as semantics. These languages encompass various types, including programming languages, data exchange languages, and domain-specific languages. Formally defining syntax and semantics, ensures that there is no mismatch between the intention of the writer, and the computer's interpretation. Context-free Grammars (CFGs) are instrumental for precisely defining the syntax of many formal languages we create. The grammars consists of a finite set of rules that is capable of generating any of the phrases that are part of an often infinite language. Moreover, computers can use these grammars to deconstruct the user's input phrases into computer-interpretable components, through a process called parsing.

Syntax highlighting is the act of assigning colors to words within sentences of a language, based on their respective roles. This is a common feature of Integrated Development Environments (IDEs), designed to assist users in working with formal languages. IDEs divide syntax highlighting into two steps: First, characters of the user input are grouped into tokens (tokenizing), afterwards, each token is assigned a color according to its type. The lexers used by IDEs, also referred to as tokenizers, are generalized and thus take grammars as part of their input. We will refer to these grammars as Lexing Grammars (LGs). Given a LG and a user input phrase, a lexer produces a token stream as output. Although certain standards exist [1], different IDEs such as Sublime Text, Visual Studio Code (VSCode), and IntelliJ IDEA may rely on varying LG formats.

CFGs and LGs both serve to define certain syntactic information of a language, but they have distinct formats, capabilities, and purposes. When developing new formal languages, developers must define certain aspects of their language in both types of grammars. The CFGs formalism is amongst other things used to facilitate parsing, while LG formats serve to enhance user interaction with their language by means of syntax highlighting. This results in syntactic concerns being spread out over various independent files, leading to increased maintenance complexity as the language evolves.

One possible way of solving this problem, is to have a single unified grammar format that specifies all information required for both parsing and syntax highlighting. A new format, however, also requires a whole new set of tools to operate on it. In order to minimize this set of tools, we would like to be able to extract CFGs and LGs in existing formats from our custom format. This way, existing tooling can be used when working with our custom grammar format.

1.2 Project Goal

In this study, we attempt to find a unified grammar format together with an approach to convert it to existing formats. We believe that an adequate format will allow us to do three things: easily specify existing CFGs in this new format without much modification, use a transformation pipeline to reach a reduced form that retains the tokenization data, easily map the reduced form to common LG formats. As such, our primary research question will be:

> *RQ 1: Can a static grammar transformation pipeline effectively be used to derive a CFG and a LG from an augmented CFG?*

We attempt to answer this question, by providing such a pipeline and measuring its effectiveness on several grammars. In order to develop such a pipeline, several intermediate questions have to be answered. First of all, we should properly understand the problem domain, and know the capabilities and limitations of common syntax highlighters. It wont be possible to consider all LG formats, but we can consider several popular formats. This forms our first intermediate research question:

> *RQ 1.1: What are the capabilities and limititions of syntax highlighters?*

[1] TextMate grammars, tree-sitter grammars

We can then start to think about developing a pipeline. The first step in this process is coming up with a grammar format for the pipeline to operate on. This format must be capable of encoding all information contained in both CFGs as well as LGs. This forms our second intermediate research question:

> *RQ 1.2: What augmentations can be made to CFGs to allow for specification of syntax highlighting concerns?*

In order to support multiple LG formats, we would like to define a common intermediate form that our pipeline targets. This form should be capable of being mapped to existing LG formats. This way we do not need to develop a separate pipeline for each format we target. This forms our third intermediate question:

> *RQ 1.3: What form of grammar could easily be mapped to various LG formats?*

Finally, we can think about the actual grammar transformations of the pipeline. Converting our initial specification grammar to a LGs will require many transformations. We would like each of these transformations to keep all tokenization concerns of the grammar in tact, such that final LG defines an equivalent tokenization for all inputs that were part of the language of the original specification. This forms our last and most important intermediate question:

> *RQ 1.4: What tokenization preserving transformations can be applied to obtain a LG?*

1.3 Research Background

The CFG format considered in this research is that of Rascal. Rascal is a meta-programming language with native and syntactical support for CFGs. This allows us to both write our pipeline in Rascal, and operate on Rascal's CFGs format. Rascal's grammars operate on source-code characters directly, a departure from the approach taken by many parser generators, which separate lexing concerns from CFG concerns. This is known as scannerless parsing, and has been studied extensively [1, 2].

Character-level grammar specification is not unique to Rascal, the formal model of CFGs also allows for this, but the resulting grammars often become either ambiguous or complex. Ambiguity is primarily related to the semantics mapped onto sentences in a language, meaning multiple interpretations are possible for a given sentence according to the specification. There are two lexical concerns that can easily result in ambiguity:

1. Many languages contain both identifiers (such as variables) and keywords, which are both sequences of alphanumeric characters
2. Identifiers do not have a fixed length

If we have a language with keywords *if* and *else*, we want to ensure that *else* is not parsed as an identifier, while *textmate* is. It is possible to use traditional CFG constructs to specify all possible identifiers except for a given set of keywords, but this results in large and hard to read definitions. One has to essentially encode a binary decision tree to specifically exclude a certain set of keywords. Similarly if we have a binary operator *in*, and whitespace between operators and operands is optional as is the case for many languages, *valueinset* could both be parsed as the binary operator *in* operating on *value* and *set*, or as the single identifier *valueinset*.

These issues are prevented by tokenizers by the following behaviors respectively:

1. If multiple regular expressions match, the one with highest priority is chosen
2. A regular expression greedily consumes as many characters as it can when matched

Rascal provides two features covering these behaviors, that can be used for disambiguating grammars: Reserve declarations, and negative follow declarations. Reserve declarations allow you to exclude a set of words from a given symbol. This way you can specify that an identifier is any non-empty alphanumeric character sequence, excluding the set of keywords. Negative follow declarations can be used to specify that a certain set of characters or words may not follow a given symbol. This can be used to forcefully match the longest possible sequence, by disallowing any character that could be included in the sequence to be matched after the sequence. This way only the longest possible match becomes valid. Rascal also supports a positive follow variant, which expects a certain sequence to follow, and both positive and negative precede variants, which expect a certain sequence to (not) precede a sequence.

1.3.1 Existing Unified Grammar Formats

Besides disambiguating features, Rascal also already provides syntax highlighting capabilities based on the same grammars that are used for parsing. Every production of a Rascal grammar can be augmented with a set of metadata attributes, which can include category strings. These categories take the place of tokens of lexing grammars. These categories are used for tokenizing, by considering the parse-tree itself. For every character leaf of the tree, the path to the root is traced and the first encountered source production that contains a category is used to assign a token to this character. This is similar to what this research is interested in, since it unifies parsing and highlighting capabilities. The approach taken by Rascal skips the intermediate LG and therefore is different from our goal with this research and comes with various drawbacks, as well as benefits.

The most noticeable drawback becomes immediately obvious when using this highlighter on changing inputs: syntactically invalid input is not highlighted at all. Since Rascal's grammars are intended for parsing, no valid parse trees are provided for syntactically incorrect inputs. This also means that no syntax highlighting based on this parse tree can be provided. Rascal simply removes all syntax highlighting whenever the input is syntactically invalid, leading to an unpleasant typing experience.

Skipping the intermediate LG also results in less tooling support. Many different approaches already exist for syntax highlighting, and certain tools heavily rely on their own tokenization approaches. As such, it might be difficult to use Rascal's highlighting within these tools. In Rascal's VSCode extension, this can be noticed first-hand. When using a grammar defined in Rascal for syntax highlighting, a slight but noticeable highlighting delay can be observed while typing, compared to the instantaneous highlighting changes when using first-party highlighting grammars. Rascal's VSCode extension relies on the semantic highlighting Application Programming Interface (API) to achieve its custom highlighting, but this approach requires more steps and is not executed synchronously like the native highlighting is. Moreover, in other IDEs or text editors it might not be possible at all to use custom highlighting approaches.

Most LGs formats are less powerful than the CFG format however. Therefore making use of the full CFGs capabilities for highlighting can be very powerful. As such our approach of attempting to generate LGs is not strictly better than using CFGs directly, but is of practical interest never the less.

Finally, it is worth mentioning that tree-sitter – a parser generator – takes the same highlighting approach as Rascal but largely removes our first drawback. tree-sitter has powerful error recovery capabilities, meaning that a parse tree is also provided for syntactically invalid input, such that highlighting continues to work as expected. This, however, does not solve the issue of lacking compatibility with other tooling.

1.4 Contributions

The relation between parsing and syntax highlighting has not been formally defined before, and there has not been any practical implementation of context-free grammar to highlighting grammar conversion either. We provide the following contributions:

TLRE Tagged Lookaround Regular Expressionss (TLREs) are a generalization of classical regular expressions, to include lookaround and subtraction operators as well as capture groups in the form of tags. Syntax and semantics for these expressions are provided in Section 3.1, together with several transformation functions in Chapter 4.

TCNFA Tagged Contextualized Non-deterministic Finite Automata (TCNFA) are a generalization of Non-deterministic Finite Automata (NFA) to capture contextual lookaround and tag information, that can serve directly as semantics for TLRE. Definitions, semantics, and a derivation from TLREs are provided in Section 3.1.4. A complete implementation of TCNFAs in Rascal is also provided, as discussed in Chapter 5.

CG Conversion Grammars (CGs) are a generalization of CFGs to serve as the bridge between CFGs and syntax highlighting grammars. These grammars support precede and follow constraints, tag assignment to non-terminal symbols, and TLREs. Syntax and semantics for these grammars are provided in Section 3.2.

LCG Lexing Conversion Grammars (LCGs) are a subset of CG that are structured like syntax highlighting grammars, specified in Section 3.3. These grammars can be mapped to various industry highlighting grammar formats such as TextMate, Ace, Monarch, and Pygments grammars.

Scope grammar Scope grammars are syntax highlighting grammars that function like TextMate grammars. Syntax and semantics for these grammars are provided in Section 4.5.1.

PDA grammar Pushdown Automaton (PDA) grammars are syntax highlighting grammars that function like Ace, Monarch, and Pygments grammars. Syntax and semantics for these grammars are provided in Section 4.5.2.

Conversion pipeline A grammar conversion pipeline is introduced in Chapter 4 that obtains TextMate, Ace, Monarch, and Pygments grammars from any CG. Three conjectures are introduced about this pipeline. Conjecture 1 states that this mapping is complete, obtaining a highlighting grammar given any CG. Conjectures 2 and 3 together relate to the soundness of the pipeline, specifying the pipeline is sound for a subset of possible CGs. An implementation of this pipeline is provided in Rascal, together with a testing and experimentation procedure, as discussed in Chapter 5.

Rascal-vis Rascal-vis is a Rascal value exploration tool introduced in Section 5.1. This tool can be used for testing and debugging of Rascal code, by giving users the ability graphically explore various types of values encountered in Rascal.

These contributions combined make it possible to construct and evaluate an automatic mapping from Rascal grammars to TextMate and Pygments highlighters. We have left a number of proof obligations to future work, in favor of empirically validating (testing) our approach on real language constructs.

1.5 Thesis Outline

Our research questions provide a good structure for tackling our problem. This structure is largely reflected in the chapters of this report:

- In Chapter 2 we analyze a couple of highlighting technologies that are popular in industry, in order to determine their capabilities and limitations.
- In Chapter 3 we introduce our own CFG format that supports encoding of highlighting information. Before discussing this format, a custom regular expression formalism is introduced to support it. Finally, we briefly discuss how our new formalism relates to the explored LG formats.
- In Chapter 4 a grammar transformation pipeline is introduced, together with a final mapping step resulting in a lexing grammar.
- In Chapter 5 we describe the technical setup for the empirical evaluation in Chapter 6.
- In Chapter 6 the pipeline is tested on various grammars, and the results are discussed.
- In Chapter 7 we summarize the results of this thesis, and provide suggestions for future research.

2 Highlighters

In this chapter we will analyze various highlighters used in industry, in order to determine what we need to achieve with our grammar transformation pipeline. This analysis is based on the combination of available documentation, and reverse engineering by developing and testing grammars. We will consider the following lexing grammar formats:

- TextMate grammars: A format introduced by the text editor TextMate, which by now is a format supported by various other text editors and IDEs.
- Monarch grammars: A format introduced by VSCode for syntax highlighting before they switched to TextMate grammars. These grammars are still used by Monaco for syntax highlighting on the web.
- Ace grammars: A format introduced by the Ace editor for syntax highlighting in their web-based text editor.
- Pygments regex grammars: A format introduced by Pygments for syntax highlighting for various output formats, which is also used by the syntax highlighter Minted for LaTeX.

We will first give an overview of each of the formats, and then discuss their similarities and differences. Before getting into the different formats, it is important clarify what the purpose of these lexing grammars exactly is. Tokenizers typically group input characters together into distinct tokens to simplify the input. In order for these tokens to be used for syntax highlighting, information is needed for what characters a given token is generated from. As such, syntax highlighters typically output sequences of tokens where each token contains the token type, and the start and end position of the token in relation to the input text. Within this report, we abstract away from the exact interface, and instead act as if the output is a token type sequence, where each character in the input is mapped to a token type. This means that the number of token types – from now on referred to as tokens – in the output is always equal to the number of characters of the input.

2.1 TextMate

TextMate's grammars have become somewhat of an industry standard for syntax highlighting in IDEs. They are supported by IDEs and code editors such as VSCode, Sublime Text and of course TextMate itself. Documentation for TextMate's grammar format is available, but does not cover all intricacies [3].

TextMate's tokenization interface is more complex than most tokenizers intended for syntax highlighting. The output of this tokenization is not a sequence of atomic tokens taking the place of each character in the input. Instead each character is assigned a so called "scope". A scope is a sequence of categories, where every category is a sequence of category components. Both of these sequences represent hierarchies. The top-level sequence, the sequence of categories, represents a dynamic hierarchy in the user input being tokenized. Consider the value `true` in the fragment `{{true}}`, this could be tokenized using the scope *block*, *block*, *boolean*. Here *block* and *boolean* are categories, and a hierarchical structure of the user input is followed. This structure can be used by highlighting themes to highlight differently depending on the context that a category appears in. The hierarchy within a category represents a hierarchical classification nature of different code-constructs. Instead of simply using the category *boolean*, we could use *constant.boolean*. This emphasizes that a boolean belongs to the parent class of constants. This hierarchy within category definitions does not play a role within the tokenization process, and is something we can largely ignore. This is merely used for providing better themes, where we can for instance target all categories belonging to *constant*, even ones that we are not aware exist. This is for instance useful when a custom language has a novel type of constant. They can use the category *constant.myConstant* that ensures their new constant is highlighted as a constant by exist themes, while allowing for custom themes to highlight this specific constant type differently from the other constants.

The grammar format of TextMate is quite declarative and focusses on the grammars intention, rather than how it is used by the syntax highlighter's internal algorithm. This format is rather minimal compared to many other formats, but is too much to discuss here nevertheless. We will focus on the most important aspects of the format. A grammar provides a list of patterns, as well as a repository. The repository is a

mapping of custom names to patterns, which can then be referred to by this name. We can distinguish two types of matching-patterns: simple patterns, and hierarchical patterns. The simple pattern specifies a regular expression – called *match* – and a map of captures. This map of captures is a mapping from numeric identifiers – representing capture groups – to categories. If the regular expression of such a pattern matches, the characters captured by each of the capture groups are assigned the category as indicated by the captures map. Since capture groups can be nested, multiple categories might be assigned to the same character. This is not a problem, since the output of this lexer is a scope per character, rather than a single category. The order of the categories follows the nesting of the captures, where a deeper nesting corresponds to occurring later in the scope. The hierarchical pattern provides two regular expressions, *begin* and *end*, each with their corresponding map of captures. They also specify a list of patterns – called *patterns* – and optionally a content name. The *begin* regular expression and map of captures behaves the same as for the simple rule, but after this rule is matched the context for processing the rest of the input changes. Now *patterns* is used to process the remainder of the input, until the *end* expression can be matched. When the *end* expression is matched, its categories are assigned similar to the *begin* match, and the rest of the input is processed according to the original list of patterns. When the patterns of this hierarchical rule are used, the provided content name is prefixed to the scopes assigned by these rules, leading to the hierarchical structure in the output.

Besides matching patterns, we can also identify two important types of structural patterns: inclusion patterns, and list patterns. Inclusion patterns simply refer to another pattern by name as defined in the repository. This allows grammars to have recursive definitions, similar to those found in CFGs. List patterns are simply patterns that include a list of other patterns. They behave the same as if the outer list that they occur in (potentially using include patterns) were flattened. These list patterns allow a single pattern name in the repository, to refer to multiple patterns at once. Grammar 1 shows an example TextMate grammar encoded in JavaScript Object Notation (JSON).

```
{
    "name": "scoped-booleans",
    "scopeName": "source.scoped-booleans",
    "patterns": [{"include": "#scope"}],
    "repository": {
        "scope": {
            "begin": "(\\{)",
            "beginCaptures": {"1": {"name": "open"}},
            "end": "(\\})",
            "endCaptures": {"1": {"name": "close"}},
            "contentName": "block",
            "patterns": [{"include": "all"}]
        },
        "boolean": {
            "match": "(true)|(false)",
            "captures": {
                "1": {"name": "boolean.true"},
                "2": {"name": "boolean.false"}
            }
        },
        "all": {"patterns": [{"include": "#boolean"}, {"include": "#scope"}]}
    }
}
```

Grammar 1: Scoped Boolean TextMate

Merely understanding the documented format is not enough, it is important to understand the accom-

panying semantics. The semantics are mostly implied in the documentation, but are not well defined. We figured out the most important details by experimenting with several grammars. When tokenizing an input, the syntax highlighter tracks the active patterns and its current position in the input. It starts with the patterns defined at the top-level, and start at the first row and column. From the current list of patterns, it checks whether any of them apply to current column of the given row. Only the characters on this line are considered, it is not able to look past linefeed character boundaries, and the linefeed boundary character of the current line is present at the end of the line. Within the list of patterns, the first one that applies at the current column is chosen. If no patterns match, the current column is skipped. When a regular expression matches, all captures are assigned the appropriate corresponding scope, and the lexer jumps to the end of the overall match to start its next token from. A regular expression can sometimes match multiple different lengths of sequences starting at the given column, in which case one is greedily chosen. This applies to all LG discussed from here on out. Inclusion and list patterns are simply substituted and flattened out when considering the patterns to apply. After an hierarchical pattern is matched it becomes active and the *end* pattern is checked against before any of the active patterns are checked. This means that the *end* expression has highest priority. An extra parameter can be used to change this to the lowest priority however, on a per pattern basis. Only the *end* expression of the pattern whose sub-patterns are currently active will be checked. If a hierarchical pattern X is active and occurred within another hierarchical pattern Y, Y's *end* expression is not checked until X becomes inactive after having matched its *end* expression. The active patterns form a stack, meaning that one pattern definition can be active multiple times in different places in this stack. This stack also tracks the corresponding content names, in order to form appropriate scopes. Fragment 1 shows a per character tokenization of {{true}}true{false} encoded in JSON (with comments), according to Grammar 1.

This tokenization process is largely deterministic. Applying patterns is done greedily, and the highlighter never reconsiders its choices. This means that if multiple patterns are applicable at once, but one leads to token being skipped later on while the other does not, the tokenizer chooses a pattern purely based on definition order and might end up skipping tokens. The only aspect of the tokenizer that is not deterministic, is regular expression matching. Within a regular expression, backtracking or other means of dealing with non-determinism takes place.

```
[
    ["open"],                          // {
    ["block", "open"],                 // {
    ["block", "block", "boolean.true"], // t
    ["block", "block", "boolean.true"], // r
    ["block", "block", "boolean.true"], // u
    ["block", "block", "boolean.true"], // e
    ["block", "close"],                // }
    ["close"],                         // }
    [],                                // t
    [],                                // r
    [],                                // u
    [],                                // e
    ["open"],                          // {
    ["block", "boolean.false"],        // f
    ["block", "boolean.false"],        // a
    ["block", "boolean.false"],        // l
    ["block", "boolean.false"],        // s
    ["block", "boolean.false"],        // e
    ["close"]                          // }
]
```

Fragment 1: Boolean Tokenization

A notable observation is that Oniguruma regular expressions are used, which feature unrestricted positive and negative lookarounds [4]. These lookarounds can be used to define multiple patterns matching the same text, without having them apply for the same input. The surrounding characters of the captured text can be used to differentiate between these patterns. This can help to deal with the restrictions resulting from the tokenizer's deterministic nature. The only limitation in this, is that these lookarounds are not able to cross line boundaries either. But the lookarounds can be used to perform empty matches in *begin* or *end* expressions, which only activate in specific contexts. This allows multiple hierarchical patterns to be exited at once, by using non-capturing regular expressions.

2.2 Monarch

Monarch grammars have similar capabilities to TextMate grammars, but the interface of its tokenizer is simpler. Each character is merely assigned a single category, rather than a scope. This category again consists of multiple components. Theming is done based on Cascading Style Sheets (CSS), and each category component is used as a class name that can be targeted by these style sheets. This means that the hierarchy of category components is not based on the ordering of components themselves, but is instead determined by the selectors in the style sheets. By using categories directly instead of scopes, these grammars can not retain the hierarchy of user input in their tokenization.

Monarch's format is a little less declarative, and reveals more about the inner workings of the tokenizer. it is also more extensive than TextMate's format, and provides several features not found in TextMate, as described in its documentation [5]. It for instance allows for more dynamic behavior, making use of capture group data. After a capture, this capture data can be referenced in various places, including assigning categories, switching states, and starting embedded tokenizers. These advanced capabilities as well as various others won't be discussed here, since they do not have an obvious counterpart in CFGs. Moreover, despite these grammars being specified in JavaScript, they do not allow dynamic JavaScript callbacks to be used during tokenization. This prevents us from using these advanced features for simulating exact CFG semantics.

Grammars in Monarch's format have to specify a map of states, together with a reference to the initial state. A state has a name that can be referred to, and consists of a list of patterns. We can distinguish 4 types of matching patterns: simple patterns, push patterns, pop patterns, and switch patterns. Each of these patterns features a regular expression of text to match, together with a list of categories to assign to each of the matched capture groups. Due to these grammars outputting categories rather than scopes, only a single capture group may apply to a given character. Moreover, this format requires every character matched by the regular expression to occur in exactly one capture group, and requires each capture group in the expression to be matched. Hence it is not possible to provide an alternation of capture groups, since only one of these capture groups would be matched for one given input. These are very strict constraints on the capture groups of regular expressions, that are not required by TextMate. Besides these expressions and categories, a push pattern specifies a state name to push to the stack, a pop pattern specifies to pop a state from the stack, and the switch pattern specifies a state to replace the top of the stack by. This grammar format also features inclusion patterns, which allow the list of patterns of one state to be included in the list of patterns of another state by reference. This once again allows for recursive definitions, similar to those seen in CFGs. Grammar 2 illustrates a Monarch grammar encoded in JavaScript, specifying similar tokenizations to the ones in Grammar 1.

```
{
    tokenizer: {
        open: [
            [/\{/, {token: "open", next: "@all"}]
        ],
        all: [
            {include: "@open"},
            [/\}/, {token: "close", next: "@pop"}],
            [/true/, {token: "boolean.true"}],
            [/false/, {token: "boolean.false"}]
        ]
    },
    start: "open",
    includeLF: true
}
```

Grammar 2: Scoped Boolean Monarch

The semantics of these grammars are easier to figure out than those of TextMate grammars. The structure of the grammar already implies the use of a stack of states. These tokenizers in fact very closely resemble single state Deterministic Push Down Automata (DPDAs) both in how they function and in use of terminology. The tokenizer contains a stack of state names, where the top of the stack represents the state the tokenizer is currently in. This stack is initialized to only contain the initial state. Once again, the tokenizer tracks the current position in the input, which is initialized to be the first row and column. For the given state, all patterns are checked in order, and the first one that matches is applied. If no patterns match, this column is skipped. Once again, patterns are matched only on the currently active line. In order to include the linefeed character to match against at the end of the line, a special flag "includeLF" has to be provided in the grammar. Moreover, only the text starting at the current column is matched against. This means that lookbehinds can not be used to differentiate between patterns based on context. Inclusion patterns are simply substituted out in the list of applicable patterns. If the applied pattern is a push pattern, the provided state will be pushed to the stack, if it is a pop pattern the top state is popped, and if it is a switch pattern a pop is performed followed by a push of the provided state. Once again, the tokenizer is fully deterministic with the exception of checking whether a regular expression applies. Fragment 2 shows a per character tokenization of `{{true}}true{false}` encoded in JSON (with comments), according to Grammar 2.

```
[
    "open",          // {
    "open",          // {
    "boolean.true",  // t
    "boolean.true",  // r
    "boolean.true",  // u
    "boolean.true",  // e
    "close",         // }
    "close",         // }
    "",              // t
    "",              // r
    "",              // u
    "",              // e
    "open",          // {
    "boolean.false", // f
    "boolean.false", // a
    "boolean.false", // l
    "boolean.false", // s
    "boolean.false", // e
    "close"          // }
]
```

Fragment 2: Boolean Tokenization Categories

Even though the patterns we discussed are the main intended patterns, due to the exact format used it is possible to combine these types. The push/pop/switch states can be provided per capture group of the regular expression. This means that arbitrary sequences of pushes and pops can be used, by adding any number of capture groups – matching a sequence of zero characters – to the end of the intended regular expression. Fragment 3 shows how this can be used to make a closing bracket pop three states from the stack.

```
[/(\})()()/, [{token: "close", next: "@pop"}, {token: "close", next: "@pop"},
↪  {token: "close", next: "@pop"}]],
```

Fragment 3: Multiple Pops Pattern Monarch

2.3 Ace

Ace grammars are very similar to Monarch grammars. They are based on DPDAs and output categories whose components are used as class names that are highlighted using CSS. Despite the documentation making no reference of state pushing and popping, these features are supported [6]. We will not explain these grammars from scratch, but instead go over important similarities and differences with Monarch grammars. These grammars do not specify an initial-state, and instead requires the state collection to contain a state called ”start”. Just like Monarch grammars, the underlying format which the grammars are defined in is JavaScript. For Monarch grammars this is not very important, but Ace allows users to define callbacks for when a pattern is matched. These callbacks allow the grammar to interact with the state of the tokenizer. The format has no dedicated syntax to push and pop multiple states at once, but this and much more can be achieved by making use of these callbacks. The restrictions on regular expressions are also less strict than Monarch’s, dropping the requirement of every capture group having to be matched. The requirement of every matched character being present in a capture group also applies to these grammars however. Inclusion patterns are not a core feature either, but can be provided and are subsequently factored

out using a built-in function call. This will simply perform substitutions before the tokenizer makes use of the grammar. Grammar 3 illustrates a Monarch grammar encoded in JavaScript, specifying the exact same tokenizations as the ones in Grammar 2.

```
const BooleanGrammar = function() {
    this.$rules = {
        start: [{regex: /\{/, token: "open", push: "all"}],
        all: [
            {include: "start"},
            {regex: /\}/, token: "close", next: "pop"},
            {regex: /(true)|(false)/, token: ["boolean.true", "boolean.false"]}
        ]
    };
    this.normalizeRules();
};
```

Grammar 3: Scoped Boolean Ace

Just like Monarch grammars, regular expressions are matched line by line. The linefeed characters are not included in this in these lines. Unlike Monarch grammars, there is no option to add linefeed characters this. When checking if a regular expression matches, the start of the line is included for context, just like TextMate does.

2.4 Pygments

Pygments allows for syntax highlighting according to any Python class that implements the appropriate lexing interface, as described by their documentation [7]. This interface expects the lexer to output one category per character, where each category is a list of category components. Like TextMate grammars, the hierarchical structure of these components is determined by the order of components themselves. Most of Pygments lexers are defined using their helper "RegexLexer" class, which uses a declarative grammar for tokenization. These grammars are once again very similar to Monarch grammars, and are based on DPDAs. Once again we will only cover some important similarities and differences between this format and previously discussed formats. Just like Ace grammars, no initial state can be specified, and instead the state collection has to contain a state called "root" to represent the initial state. One important difference with any of the previously discussed tokenizers, is that this tokenizer does not operate on a single line at a time. A regular expression is allowed to cross line boundaries. This format also supports pushing and popping of multiple states at a time, in the form of a dedicated syntactic feature. Grammar 4 illustrates a Monarch grammar encoded in JavaScript, specifying the same tokenizations as the ones in Grammar 2.

```
class BooleanLexer(RegexLexer):
    tokens = {
        'root': [include('open')],
        'open': [(r'\{', Token.Open, 'all')],
        'all': [
            include('open'),
            (r'\}', Token.Close, '#pop'),
            (r'(true)|(false)', bygroups(Token.Boolean.True, Token.Boolean.False))
        ]
    }
```

Grammar 4: Scoped Boolean Pygments

2.5 Lexing Grammar Similarities

Many of these formats appear to be DPDA based and follow similar structures. TextMate is the one exception, but is also the most common format amongst them. TextMate's tokenization interface is the most powerful amongst the discussed formats, but its language recognition capabilities is the weakest. The hierarchical patterns of TextMate can be simulated by the DPDA based grammars as long as no content name is present. This is done by using a push pattern of the *begin* expression of the pattern, which pushes a new state that includes a list of patterns representing TextMate *patterns*, with a pop pattern of the *end* expression inserted at the start of these patterns. Simulating DPDA based grammars using a TextMate grammar is not as obvious. The behavior of pushing and popping a combination of states as seen in DPDA based grammars has no obvious counterpart in TextMate grammars.

We will use the discussed capabilities of TextMate grammars as our target. This will ensure that in terms of recognition, all other formats that we are interested in can be derived as well. In the process of deriving DPDA based grammars, we might lose some information of the exact assigned scopes however, since they have to be transformed into single categories.

3 Formalizations

We now know what our source format is and understand the capabilities of our target format. These formats are quite different, especially regarding the aspect of regular expressions. Rascal's CFG source format does not support regular expressions with tags, while this forms a crucial aspect of LGs. For this reason, we introduce our own CG format. This format combines the capabilities of both formats. This allows us to easily map a Rascal grammar to a CG, apply several transformation rules to get rid of constructs that are not supported by the target grammar format, and finally map the resulting CG to the target LG format.

This CG format could also serve as our unified grammar format, containing both tokenization and parsing information. From a grammar in this format, a CFG in Rascal's format could be derived using a mapping scheme. For implementation purposes, we will consider Rascal's format as the unified source format, since Rascal has dedicated syntax built in to their language for defining CFGs in this format. This makes it convenient to stick with their format for specifying grammars.

In this chapter, we will introduce this CG format, including a formalization. In order to support this format, we will first formalize the notion of regular expressions that include lookarounds and capture-groups. Finally we will discuss the relation between our grammar format, Rascal's CFG format, and LG formats.

3.1 Regular Expressions

Regular expressions play an important role in all of the LGs formats that we have considered. The theory of regular expressions is well defined and researched after Kleene introduced Regular Languages in 1951 [8]. Unfortunately, there is a disconnect between the basic formalism of regular expressions, and the capabilities commonly found in tools that support regular expressions for pattern matching. Notably, the classical formalism does not support lookaround operators such as lookaheads and lookbehinds, and has no notion of capture groups. Formal regular expressions were originally intended to define infinite languages using finite syntax. Within this context, there simply is no obvious notion of lookaround operators and capture groups. In regular expression tools, lookarounds are used to match phrases in the language only if they appear in an appropriate larger context. This is for instance used when performing searches. A positive lookahead could for instance specify to only match (and return) a certain pattern, if it is followed by another pattern. Capture groups are used to split a pattern match into distinct parts, and retrieve the matched text for each part individually. In order to support these features within a formal framework, we have to alter our notion of languages.

We will first consider a formalization of regular expressions with lookarounds and a separate formalization with capture groups, before combining the two. There is an interplay between these concepts, but looking at them in isolation helps to properly understand the idea. Finally we consider how to reason with regular expressions defined in our format, by introducing an automaton formalism.

3.1.1 Lookaround Regular Expressions

Regular expressions with lookaheads have received more attention in recent years, and have been researched most notably by Takayuki Miyazaki *et al.* and by Martin Berglund *et al.* [9, 10]. This research does however not include lookbehinds, and can not trivially be extended to include lookbehinds. We will therefore define our own notion of lookarounds. This notion shares several similarities to these works, but is syntactically inspired by Rascal's precede and follow operators.

In order to add lookarounds to regular expressions, we have to make sure there is a notion of context in the corresponding languages. A regular expression defined over the alphabet Σ typically defines a language that is a subset of Σ^*. Here X^* denotes the Kleene closure of set X. It represents the smallest set that contains the empty string ϵ and all sequences of one or more symbols that exist in X: $X^* = \{\epsilon\} \cup X \cup X^2 \cup X^3 \cup \ ... \ .$ The language that our regular expression defines will instead be a subset of $\Sigma^* \times \Sigma^* \times \Sigma^*$. The entries in this language consist of three parts:

- The prefix that is allowed before the body
- The body that is matched

- The suffix that is allowed after the body

These prefix and suffix parts form the context of the match. When relating this to a search problem, one could consider the concatenation of all three character sequences as the text that was searched in, and the *body* as the match that was found by the regular expression. For example, the tuple (a, b, c) would represent that b was found in the overall input of abc. In addition to lookarounds, we will also add the notion of subtraction to our regular expressions.

Lookaround regular expressions can be defined inductively, according to grammar

$$R ::= \mathbf{0} \mid \mathbf{1} \mid \epsilon \mid a \mid R{+}R \mid RR \mid R^{+} \mid R{>}R \mid R{<}R \mid R{\not>}R \mid R{\not<}R \mid R{-}R \mid (R)$$

for every $a \in \Sigma$. The expression $r_1{>}r_2$ represents a positive lookahead, expressing that a match of r_1 should be followed by a match of r_2 without r_2 being included in the overall match, as it would be for concatenation. Meanwhile $r_1{\not>}r_2$ represents a negative lookahead, expressing that a match of r_1 may not be follow by a match of r_2. $r_2{<}r_1$ and $r_2{\not<}r_1$ represent a lookbehind and negative lookbehind respectively, representing r_1 should (not) be preceded by r_2. We define the precedence levels and associativity as follows:

1. $\mathbf{0} \mid \mathbf{1} \mid \epsilon \mid a \mid (R)$
2. R^{+}
3. RR which is left-associative
4. $R{+}R$ which is associative
5. $R{-}R$ which is right-associative
6. $R{>}R \mid R{\not>}R$ which are right-associative
7. $R{<}R \mid R{\not<}R$ which are left-associative

In this list, the base cases numbered 1 have the highest precedence, while the lookbehinds have the lowest precedence. Alternation $R{+}R$ being associative can be confirmed by the semantics we will now assign.

The exact language of a given lookaround regular expression r can then be derived by following its inductive structure. Let r_1 and r_2 be two regular expressions which possibly make up the expression of r. The language for any possible shape of r is defined as follows:

$$\begin{aligned}
\mathcal{L}(\mathbf{0}) &= \emptyset \\
\mathcal{L}(\mathbf{1}) &= \{(p, w, s) \mid p, w, s \in \Sigma^*\} \\
\mathcal{L}(\epsilon) &= \{(p, \epsilon, s) \mid p, s \in \Sigma^*\} \\
\mathcal{L}(a) &= \{(p, a, s) \mid p, s \in \Sigma^*\} \\
\mathcal{L}(r_1{+}r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 r_2) &= \{(p, \alpha\beta, s) \mid (p, \alpha, \beta s) \in \mathcal{L}(r_1) \wedge (p\alpha, \beta, s) \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1^{+}) &= \cup_{i \geq 1} S_i(r_1) \\
\mathcal{L}(r_1{>}r_2) &= \{(p, w, s) \in \mathcal{L}(r_1) \mid \exists\, \alpha, \beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw, \alpha, \beta) \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1{\not>}r_2) &= \{(p, w, s) \in \mathcal{L}(r_1) \mid \neg\exists\, \alpha, \beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw, \alpha, \beta) \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1{<}r_2) &= \{(p, w, s) \in \mathcal{L}(r_2) \mid \exists\, \alpha, \beta \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha, \beta, ws) \in \mathcal{L}(r_1)\} \\
\mathcal{L}(r_1{\not<}r_2) &= \{(p, w, s) \in \mathcal{L}(r_2) \mid \neg\exists\, \alpha, \beta \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha, \beta, ws) \in \mathcal{L}(r_1)\} \\
\mathcal{L}(r_1{-}r_2) &= \mathcal{L}(r_1) \backslash \mathcal{L}(r_2) \\
\mathcal{L}((r_1)) &= \mathcal{L}(r_1)
\end{aligned}$$

We define $S_i(r_1)$ for a lookaround regular expression r_1 inductively as the sequence of i elements in the language of r_1 :

$$\begin{aligned}
S_0(r_1) &= \mathcal{L}(\epsilon) = \{(p, \epsilon, s) \mid p, s \in \Sigma^*\} \\
S_{i+1}(r_1) &= \{(p, \alpha\beta, s) \mid (p, \alpha, \beta s) \in S_i(r_1) \wedge (p\alpha, \beta, s) \in \mathcal{L}(r_1)\}
\end{aligned}$$

Note that our base expressions $\mathbf{1}$, ϵ and a all define languages that include all possible prefixes and suffixes paired with their respective body. This means that their respective bodies can be matched within any context. Similarly, the concatenation and Kleene star operations preserve all prefixes and suffixes, but they also ensure that the the combined bodies are indeed allowed by the respective prefix or suffix they are combined with. More specifically, concatenation only concatenates a triplet x with another triplet y, if the start of the suffix of x matches the body of y, and symmetrically the end of the prefix of y matches the body of x. If lookarounds were not added, this would essentially leave us with a more complex formulation of the same semantics as is present in classical regular expressions. The extra notions of lookarounds and subtractions are not used to define new entries in the language like the classical regular expression operators do, instead they serve to filter out some entries present in the language defined by a sub-expression.

3.1.2 Tagged Regular Expressions

Next we consider how to deal with capture groups. We do this by tagging characters of words in the language with the identifier of a given capture group. This way the language itself encodes what character sequences belong to which capture group. In order to achieve this, we introduce the concept of tag groups into the syntax, which can then be used as capture groups if every tag group specifies a unique tag. Moreover, in this thesis we are interested in assigning categories to characters, which tags can be used for directly, skipping the intermediate capture group structure. These regular expressions are expressions over an alphabet Σ and a tag universe T. The resulting language is a subset of $(\Sigma \times \mathcal{P}(T))^*$. Note that every character in words of our language is now paired with a subset of the tag-universe, rather than exactly 1 tag.

The idea of encoding capture groups in regular expressions using tags is not new. Ville Laurikari used tags to efficiently encode capture groups, and came up with the notions of tagged (non-)deterministic automata to efficiently match such regular expressions [11]. Their notion of tags does however differ from ours, and focusses on regular expression matching, rather than language analysis.

Tagged regular expressions of this format can once again be defined inductively, according to grammar

$$R ::= \mathbf{0} \mid \epsilon \mid a \mid R{+}R \mid RR \mid R^+ \mid (\langle t \rangle R) \mid (R)$$

for every $a \in \Sigma$ and $t \in T$. The precedences of operators here are the same as specified in the previous section, with $(\langle t \rangle R)$ belonging to the base-case group with the highest precedence.

Again, the exact language of a given tagged regular expression r can then be derived by following its inductive structure. Let r_1 and r_2 be two regular expressions which possibly make up the expression of r. The language for any possible shape of r is defined as follows:

$$\begin{aligned}
\mathcal{L}(\mathbf{0}) &= \emptyset \\
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(a) &= \{(a, \emptyset)\} \\
\mathcal{L}(r_1{+}r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 r_2) &= \{\alpha\beta \mid \alpha \in \mathcal{L}(r_1) \wedge \beta \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1^+) &= \cup_{i \geq 1} S_i(r_1) \\
\mathcal{L}((\langle t \rangle r_1)) &= \{(w, \{t\} \cup T') \mid (w, T') \in \mathcal{L}(r_1)\} \\
\mathcal{L}((r_1)) &= \mathcal{L}(r_1)
\end{aligned}$$

We define $S_i(r_1)$ for a tagged regular expression r_1 inductively as the sequence of i elements in the language of r_1 :

$$\begin{aligned}
S_0(r_1) &= \mathcal{L}(\epsilon) = \{\epsilon\} \\
S_{i+1}(r_1) &= \{\alpha\beta \mid \alpha \in S_i(r_1) \wedge \beta \in \mathcal{L}(r_1)\}
\end{aligned}$$

The set of tags is not actively considered by any expression constructs, except for the operator that assigns tags. This tag assignment operator augments the words in the language defined by its sub-expression, with the specified tag.

Note that we implicitly switch the form of the main body from $(\Sigma \times T^*)^*$ to $\Sigma^* \times (T^*)^*$ in the tag operator language definition. This makes it easier to separate the character sequence from the scope sequence, but there still is a one to one correspondence between characters and scopes which is not immediately clear in this notation. This implicit conversion will be used in multiple places. Additionally, the union notation is not used entirely properly either. It specifies that the union between a tag set sequence T' and a single tag set t is taken, while intending to represent taking the union with t for every tag set in the sequence of T'.

In the language defined by a tagged regular expression, a given character sequence might occur multiple times, with different tags associated per character. Regular expressions and languages for which such a character sequence exists, are said to be tag-ambiguous.

3.1.3 Tagged Lookaround Regular Expressions

Now that we understand the concepts of tags and lookarounds in isolation, we can merge these notions to define tagged lookaround regular expressions. For a given alphabet Σ and tag universe T, we declare $\mathcal{R}(\Sigma, T)$ to be the universe of tagged lookaround regular expressions, inductively defined according to grammar:

$$R ::= \mathbf{0} \mid \mathbf{1} \mid \epsilon \mid a \mid R{+}R \mid RR \mid R^+ \mid R{>}R \mid R{<}R \mid R{\not>}R \mid R{\not<}R \mid R{-}R \mid (\langle t \rangle R) \mid (R)$$

for every $a \in \Sigma$ and $t \in T$. The precedences for these operators are the same as specified in the previous sections. In order to make some common structures easier to write, we can also introduce some additional shorthand notation:

- $(>R) = \epsilon{>}R$
- $(R<) = R{<}\epsilon$
- $(\not>R) = \epsilon{\not>}R$
- $(R\not<) = R{\not<}\epsilon$
- $(-R) = \mathbf{1}{-}R$
- $R^* = \epsilon{+}R^+$

For the semantics of these expressions, the concepts of the previously defined expressions can largely be merged straightforwardly. The language of tagged lookaround regular expressions will be subsets of $(\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^*$. This means that the contextual parts of the language may also contain tags, and thus special care has to be taken with lookarounds and concatenation. Tags present in the contextual section of a language have to be merged with the corresponding part of the other language appropriately. Let r_1 and r_2 be two regular expressions which possibly make up the expression of r. The

language for any possible shape of r is defined as follows:

$$
\begin{aligned}
\mathcal{L}(\mathbf{0}) &= \emptyset \\
\mathcal{L}(\mathbf{1}) &= \{(p,w,s) \mid p,w,s \in (\Sigma \times \{\emptyset\})^*\} \\
\mathcal{L}(\epsilon) &= \{(p,\epsilon,s) \mid p,s \in (\Sigma \times \{\emptyset\})^*\} \\
\mathcal{L}(a) &= \{(p,(a,\emptyset),s) \mid p,s \in (\Sigma \times \{\emptyset\})^*\} \\
\mathcal{L}(r_1{+}r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 r_2) &= \{((p, T_1^p \cup T_2^p), (\alpha, T_1^\alpha \cup T_2^\alpha)(\beta, T_1^\beta \cup T_2^\beta), (s, T_1^s \cup T_2^s)) \\
&\quad \mid ((p, T_1^p), (\alpha, T_1^\alpha), (\beta, T_1^\beta)(s, T_1^s)) \in \mathcal{L}(r_1) \wedge ((p, T_2^p)(\alpha, T_2^\alpha)), (\beta, T_2^\beta)), (s, T_2^p)) \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1^+) &= \cup_{i \geq 1} S_i(r_1) \\
\mathcal{L}(r_1{>}r_2) &= \{((p, T_1^p \cup T_2^p), (w, T_1^w \cup T_2^w), (\alpha\beta, T^s \cup (T^\alpha T^\beta))) \\
&\quad \mid ((p, T_1^p), (w, T_1^w), (\alpha\beta, T^s)) \in \mathcal{L}(r_1) \wedge ((p, T_2^p)(w, T_2^w), (\alpha, T^\alpha), (\beta, T^\beta)) \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1 \not> r_2) &= \{((p, T_1^p), (w, T_1^w), (s, T^s)) \in \mathcal{L}(r_1) \\
&\quad \mid \neg\exists\, \alpha,\beta \in \Sigma^*, T_2^p, T_2^w, T^\alpha, T^\beta \subseteq T \,.\, s = \alpha\beta \wedge ((p, T_2^p)(w, T_2^w), (\alpha, T^\alpha), (\beta, T^\beta)) \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1{<}r_2) &= \{((\alpha\beta, T^p \cup (T^\alpha T^\beta)), (w, T_1^w \cup T_2^w), (s, T_1^s \cup T_2^s)) \\
&\quad \mid ((\alpha, T^\alpha), (\beta, T^\beta), (w, T_1^w)(s, T_1^s)) \in \mathcal{L}(r_1) \wedge ((\alpha\beta, T^p), (w, T_2^w), (s, T_2^s)) \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1 \not< r_2) &= \{((p, T^p), (w, T_2^w), (s, T_2^s)) \in \mathcal{L}(r_2) \\
&\quad \mid \neg\exists\, \alpha,\beta \in \Sigma^*, T^\alpha, T^\beta, T_2^w, T_2^s \subseteq T \,.\, p = \alpha\beta \wedge ((\alpha, T^\alpha), (\beta, T^\beta), (w, T_2^w)(s, T_2^s)) \in \mathcal{L}(r_1)\} \\
\mathcal{L}(r_1{-}r_2) &= \{((p, T_1^p), (w, T_1^w), (s, T_1^s)) \in \mathcal{L}(r_1) \mid \neg\exists\, T_2^p, T_2^w, T_2^s \subseteq T \,.\, (p, T_2^p), (w, T_2^w), (s, T_2^s)) \in \mathcal{L}(r_2)\} \\
\mathcal{L}((\langle t \rangle r_1)) &= \{(p, (w, \{t\} \cup T^w), s) \mid (p, (w, T^w), s) \in \mathcal{L}(r_1)\} \\
\mathcal{L}((r_1)) &= \mathcal{L}(r_1)
\end{aligned}
$$

Again $S_i(r_1)$ for a tagged lookaround regular expression r_1 is defined inductively as the sequence of i elements in the language of r_1 :

$$
\begin{aligned}
S_0(r_1) &= \mathcal{L}(\epsilon) = \{(p,\epsilon,s) \mid p,s \in (\Sigma \times \{\emptyset\})^*\} \\
S_{i+1}(r_1) &= \{((p, T_1^p \cup T_2^p), (\alpha, T_1^\alpha \cup T_2^\alpha)(\beta, T_1^\beta \cup T_2^\beta), (s, T_1^s \cup T_2^s)) \\
&\quad \mid ((p, T_1^p), (\alpha, T_1^\alpha), (\beta, T_1^\beta)(s, T_1^s)) \in S_i(r_1) \wedge ((p, T_2^p)(\alpha, T_2^\alpha)), (\beta, T_2^\beta)), (s, T_2^p)) \in \mathcal{L}(r_1)\}
\end{aligned}
$$

These definitions have gotten a fair bit more complex, but are mostly an intuitive combination of the previously explored concept. Concatenation and positive lookarounds take care of merging the tagsets properly, while negative lookarounds and subtraction ignore the tags provided in the second expression by using an existential quantifier over all possible tags.

The tag semantics provided by our formalism are slightly different than what most practical regular expressions engines support. Firstly, using this approach we can not differentiate not matching a capture group, which happens when the capture group occurs in an alternation that was not used, or capturing an empty string. Most regex engines differentiate this behavior, by providing a null value when the capture group was not matched. Secondly, most regular expression engines only return the last capture of a capture group. Hence if a capture group occurs within an iterative construct, some of the capture data may be lost. The difference between empty captures and no capture is irrelevant for our purpose and can be ignored, the difference in behavior within iterative constructs is of importance however, and will require special care later in our transformation pipeline to ensure our theory does not yield incorrect results. The discussion provides some possibilities for solving these issues to obtain a wider applicable formalism.

3.1.4 Tagged Contextualized Non-deterministic Finite Automata

In the field of language theory, the notion of regular expression is accompanied by the concepts of NFAs and Deterministic Finite Automata (DFA) [12, 13]. All three of these formalisms can define regular languages,

and are equally powerful. Regular expressions are great for constructively building up a language in an intuitive way, but DFAs have the nice property of being easy to compute with. Given two DFAs defining languages L_1 and L_2, we can easily construct a DFA representing any of the following languages: $L_1 \cup L_2$, $L_1 \cap L_2$, or $\Sigma^* \backslash L_1$. Mosts importantly, it is trivial to check whether the language represented by a given DFA is empty. A given regular expression can be translated to a NFA with an equivalent language, which in turn can be translated into a DFA with an equivalent language.

Within the context of this research, being able to take the intersections and complements of the languages of two regular expressions is very valuable. These operations combined with a check for whether the language is empty, allow us to answer any of these questions:

- Are two given regular expressions semantically equivalent?
- Do any of the words defined by a regular expression form a prefix of words of another regular expression?
- Does a given regular expression define empty words?

All of these questions later become important questions, that should be answered in an automated way to make informed decisions during our grammar conversion process.

For this reason, we define automata with similar capabilities that support the language format of tagged lookaround regular expressions: Tagged Contextualized Deterministic Finite Automata (TCDFA) and TCNFAs. We will first introduce the TCNFAs model, and then relate it to the TCDFA model.

Our automata are defined in relation to classic NFAs. Therefore we will first formally define NFAs. A NFA is a tuple (Q, Σ, R, i, F), where:

- Q is a finite set of states
- Σ is a finite set of symbols, defining the alphabet
- $R \subseteq Q \times (\{\epsilon\} \cup \Sigma) \times Q$ is a transition relation
- $i \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states

The language of a given $N = (Q, \Sigma, R, i, F)$ is defined as follows:

$$\mathcal{L}(N) = \{w \in \Sigma^* \mid \delta(i, w) \cap F \neq \emptyset\}$$

In this definition $\delta \subseteq Q \times \Sigma^* \times \mathcal{P}(Q)$ specifies the set of reachable states for a given word. Using the reflexive transitive closure R_ϵ^* of $R_\epsilon = \{(u, v) \mid (u, \epsilon, v) \in R\}$, we can δ as follows:

$$\begin{aligned} \delta(q, \epsilon) &= \{s \mid (q, s) \in R_\epsilon^*\} \\ \delta(q, a\alpha) &= \{v \mid (q, s) \in R_\epsilon^* \wedge (s, a, u) \in R \wedge v \in \delta(u, \alpha)\} \qquad \text{for } a \in \Sigma, \alpha \in \Sigma^* \end{aligned}$$

A TCNFA is defined by a tuple $(Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$, where:

- Q_p is a finite set of prefix states
- Q_w is a finite set of main states, disjoint from Q_p
- Q_s is a finite set of suffix states, disjoint from Q_p and Q_w
- Σ is a finite set of symbols
- T is a finite set of tags
- $R \subseteq (Q_p \cup Q_w \cup Q_s) \times ((\Sigma \times \mathcal{P}(T)) \cup \{\epsilon, \langle, \rangle\}) \times (Q_p \cup Q_w \cup Q_s)$ is a transition relation
- $i \in Q_p$ is the initial state

- $F \subseteq Q_s$ is the set of accepting states

We say that $TCNFA(\Sigma, T)$ is the universe of all TCNFAs with the given alphabet and tags universe.

The language represented by a TCNFA is a subset of $(\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^*$. This language can be described in terms of the language of a NFA. Given a TCNFA $C = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$, we can define a NFA $N = (Q_p \cup Q_w \cup Q_s, (\Sigma \times \mathcal{P}(T)) \cup \{\langle, \rangle\}, R, i, F)$. The language of C is then defined as:

$$\mathcal{L}(C) = \{(p, w, s) \in (\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^* \mid p\langle w \rangle s \in \mathcal{L}(N)\}$$

A TCNFA essentially makes tags an intrinsic part of the alphabet, and uses special separation characters $\langle$ and $\rangle$ to split phrases of the language into 3 parts separating the context from the main match. We will refer to transitions matching either $\langle$ or $\rangle$ as bracket transitions.

Considering this definition of the language, stricter conditions could be added to the transition relation of a TCNFA without affecting the universe of languages that can be defined by TCNFAs. This comes from the $\langle$ and $\rangle$ symbols always having to occur exactly once in a fixed order within the language specified by the corresponding NFA. Therefore we can specify the bracket constraint, which requires R to be a subset of:

$$\begin{aligned} & Q_p \times ((\Sigma \times \mathcal{P}(T)) \cup \{\epsilon\}) \times Q_p \\ \cup\ & Q_w \times ((\Sigma \times \mathcal{P}(T)) \cup \{\epsilon\}) \times Q_w \\ \cup\ & Q_s \times ((\Sigma \times \mathcal{P}(T)) \cup \{\epsilon\}) \times Q_s \\ \cup\ & Q_p \times \{\langle\} \times Q_w \\ \cup\ & Q_w \times \{\rangle\} \times Q_s \end{aligned}$$

The universe of TCDFAs is a subset of that of TCNFAs. If a TCNFA does not contain any ϵ-transitions, and its relation is a function, it is said to be a TCDFA. These requirements alone do however not ensure that the δ-function of the corresponding NFA always reaches exactly one state, as would be the case for a DFA. This is an important property of DFAs, that allows their complement to be calculated. They achieve this by requiring totality of the transition function, but this unfortunately contradicts the bracket constraint that we just set on the transition relation of a TCNFA. Instead we will directly add the following, less trivial to achieve, totality requirement: $\forall\ p, w, s \in (\Sigma \times \mathcal{P}(T))^*\ .\ |\delta(p\langle w \rangle s)| = 1$

Converting a TCNFA C to a TCDFA C_d with an equivalent language can now be done using the standard NFA to DFA subset construction approach. For the NFA corresponding to C, we calculate a DFA D with an equivalent language. We then partition the states of D according to the paths they can be reached through. For a given state q, and every path π that it can be reached through:

- If π contains neither $\langle$ nor $\rangle$, q belongs to Q_p
- If π contains a $\langle$ but not $\rangle$, q belongs to Q_w
- If π contains both $\langle$ and $\rangle$, q belongs to Q_s

This scheme will perfectly assign each state to a single state set, except for possibly $q = \emptyset$. This is the case because all reachable states in C must be reached through a path that adheres to constraints of a TCNFA transition relation, which are designed to ensure this. The sink state $q = \emptyset$ does however not follow these path constraints. In case we do care about the determinism aspect of our TCNFA, but do not require it to be a full TCDFA, all we have to do is delete this $\emptyset$ state and all its corresponding transitions. Because this state forms a sink and is not accepting, the described language is unaffected by this modification. After deleting this sink state, the totality requirement can still be met by defining three sink states, one for each state set of the partition: $s_p \in Q_p$, $s_w \in Q_w$, and $s_s \in Q_s$. Then for a given a given transition relation R of D, with the

empty state transitions removed, we can augment R by the following transitions:

$$\begin{aligned}
&\{s \xrightarrow{(a,t)} s_p \mid s \in Q_p \wedge (a,t) \in \Sigma \times \mathcal{P}(T) \wedge \neg\exists\, v \in Q_p \,.\, s \xrightarrow{(a,t)} v \in R\} \\
\cup\ &\{s \xrightarrow{(a,t)} s_w \mid s \in Q_p \wedge (a,t) \in \Sigma \times \mathcal{P}(T) \wedge \neg\exists\, v \in Q_w \,.\, s \xrightarrow{(a,t)} v \in R\} \\
\cup\ &\{s \xrightarrow{(a,t)} s_s \mid s \in Q_p \wedge (a,t) \in \Sigma \times \mathcal{P}(T) \wedge \neg\exists\, v \in Q_s \,.\, s \xrightarrow{(a,t)} v \in R\} \\
\cup\ &\{s \xrightarrow{\langle} s_w \mid s \in Q_p \wedge \neg\exists\, v \in Q_w \,.\, s \xrightarrow{\langle} v \in R\} \\
\cup\ &\{s \xrightarrow{\rangle} s_s \mid s \in Q_w \wedge \neg\exists\, v \in Q_s \,.\, s \xrightarrow{\rangle} v \in R\}
\end{aligned}$$

In this notation, $s \xrightarrow{a} u$ simply represents a tuple (s, a, u), while hinting at this tuple representing a transition.

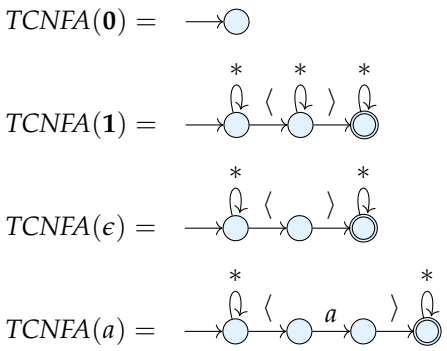

Figure 1: Base TCNFA Automata

A regular expression can be mapped to a TCNFA with an equivalent language, using an inductive definition. Given a tagged lookaround regular expression r, $TCNFA(r)$ will define the TCNFA of r, such that $\mathcal{L}(r) = \mathcal{L}(TCNFA(r))$. The base cases of the inductive definition can be defined directly:

$$TCNFA(\mathbf{0}) = (\{p\}, \emptyset, \emptyset, \Sigma, T, \emptyset, p, \emptyset)$$

$$TCNFA(\mathbf{1}) = (\{p\}, \{m\}, \{s\}, \Sigma, T, \{p \xrightarrow{\langle} m, m \xrightarrow{\rangle} s\} \cup \{p \xrightarrow{(a,\emptyset)} p, m \xrightarrow{(a,\emptyset)} m, s \xrightarrow{(a,\emptyset)} s \mid a \in \Sigma\}, p, \{s\})$$

$$TCNFA(\epsilon) = (\{p\}, \{m\}, \{s\}, \Sigma, T, \{p \xrightarrow{\langle} m, m \xrightarrow{\rangle} s\} \cup \{p \xrightarrow{(a,\emptyset)} p, s \xrightarrow{(a,\emptyset)} s \mid a \in \Sigma\}, p, \{s\})$$

$$TCNFA(a) = (\{p\}, \{m_1, m_2\}, \{s\}, \Sigma, T, \{p \xrightarrow{\langle} m_1, m_1 \xrightarrow{a} m_2, m_2 \xrightarrow{\rangle} s\} \cup \{p \xrightarrow{(a,\emptyset)} p, s \xrightarrow{(a,\emptyset)} s \mid a \in \Sigma\}, p, \{s\})$$

Figure 1 illustrates each of these automata graphically. This figure follows standard automata conventions where the double ring indicates the accepting state, and the incoming arrow without a starting state indicates the initial state. Specific to our illustration is that $*$ represents any character transition, but does not match $\langle$ or $\rangle$.

The base automata do not have to be defined in terms of other automata, and are therefore fairly straightforward. The inductively defined operators pose more of a challenge. Let r_1 and r_2 be two regular expressions, with their corresponding TCNFAs:

- $(Q_1^p, Q_1^w, Q_1^s, \Sigma, T, R_1, i_1, F_1) = TCNFA(r_1)$
- $(Q_2^p, Q_2^w, Q_2^s, \Sigma, T, R_2, i_2, F_2) = TCNFA(r_2)$

Using these sub-automata, automata of inductively defined syntax can be specified. Most of these automata take a shape very similar to product automata, essentially simulating the execution of both sub-automata in parallel, while synchronizing their actions where necessary. We can define $TCNFA(r_1 r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= Q_1^p \times Q_2^p \\
Q_w &= (Q_1^w \times Q_2^p) \cup (Q_1^s \times Q_2^w) \\
Q_s &= Q_1^s \times Q_2^s \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{\epsilon} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_2 \xrightarrow{\epsilon} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^p \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\langle} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\rangle} s_1' \in R_1 \wedge s_2 \xrightarrow{\langle} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1, s_2') \mid s_1 \in Q_1^s \wedge s_2 \in Q_2^w \wedge s_2 \xrightarrow{\rangle} s_2' \in R_2\} \\
i &= (i_1, i_2) \\
F &= F_1 \times F_2
\end{aligned}
$$

This definition essentially encodes a synchronization between the two given automata. Epsilon transitions may occur non-synchronized, but character transitions are always synchronized and their corresponding tags are merged. The match-start symbol of the first automaton may be taken at any point, and does not have to be synchronized. The match-end symbol of the first automaton does however need to be synchronized with the match-start symbol of the second automaton, and is replaced by an epsilon transition in the resulting automaton. Finally the match-end symbol of the second automaton may appear at any point. Figure 2 gives an example of how two specific TCNFAs can be used to define an automaton specifying their concatenation. Note that the initial TCNFAs that the concatenation are built-up from are simplified versions of the TCNFAs that would be obtained from our inductive definition to improve readability. Figure 3 illustrates more abstractly how the structure of the concatenation TCNFA is built-up from any two TCNFAs. In this diagram, the 3 distinct parts – prefix, match, and suffix – that any TCNFA is built-up from are highlighted. Every state in the combined automaton is formed by combining a state of both automaton as indicated by the grey connection lines. Most notably, this figure illustrates how bracket transitions are combined, for instance synchronizing the match-end transition of the first automaton with the match-start automaton of the second automaton to form an epsilon transition in the final automaton.

The definition of a lookaround TCNFA very closely resembles that of a concatenation TCNFA, but changing how synchronization on match-start and match-end symbols is performed. Figure 4 shows a specific example of how a lookahead TCNFA is constructed, and Figure 5 illustrates more abstracticly how the structure of the lookahead TCNFA is built-up from two TCNFAs.

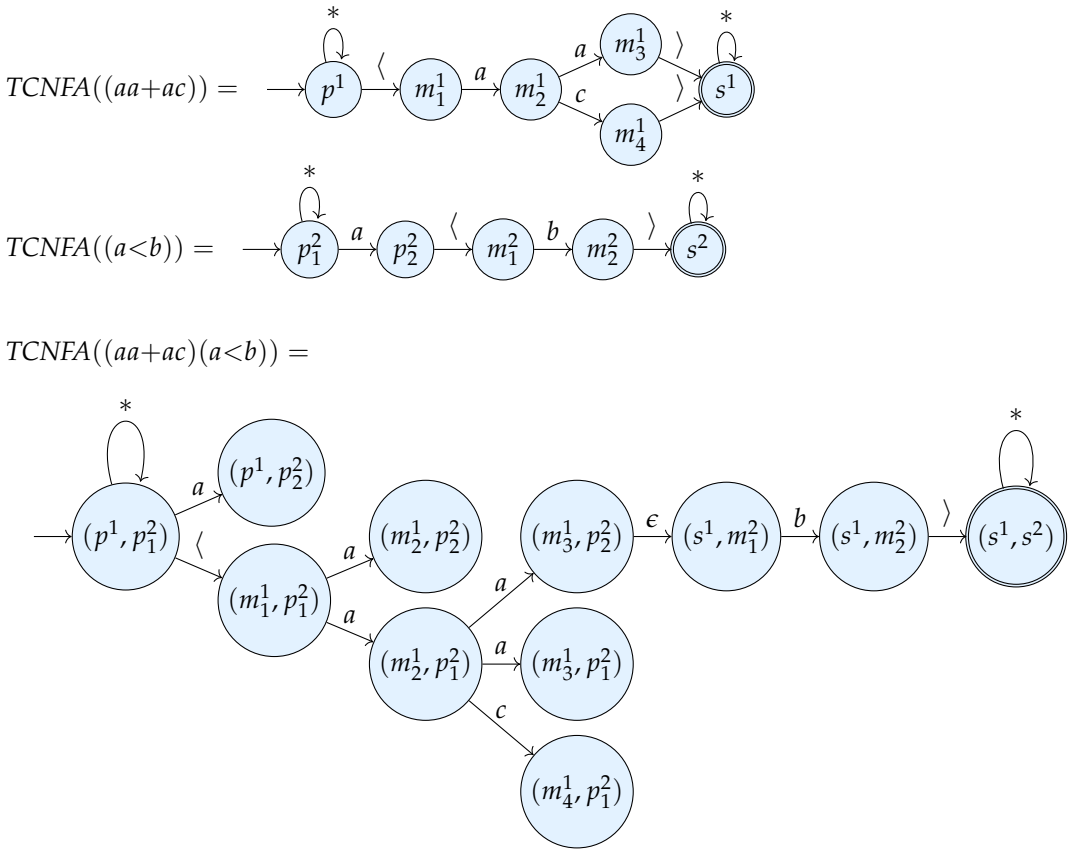

Figure 2: Concatenation TCNFA Example

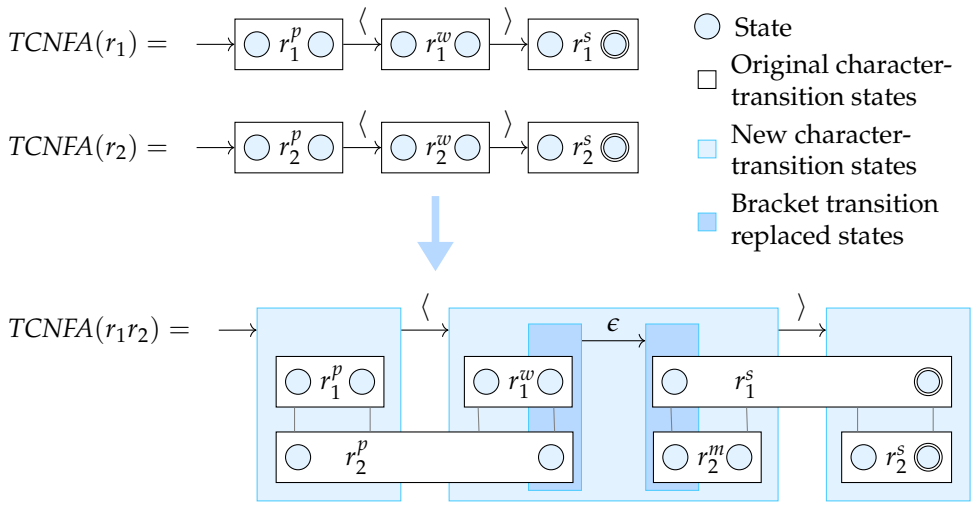

Figure 3: Concatenation TCNFA Structure

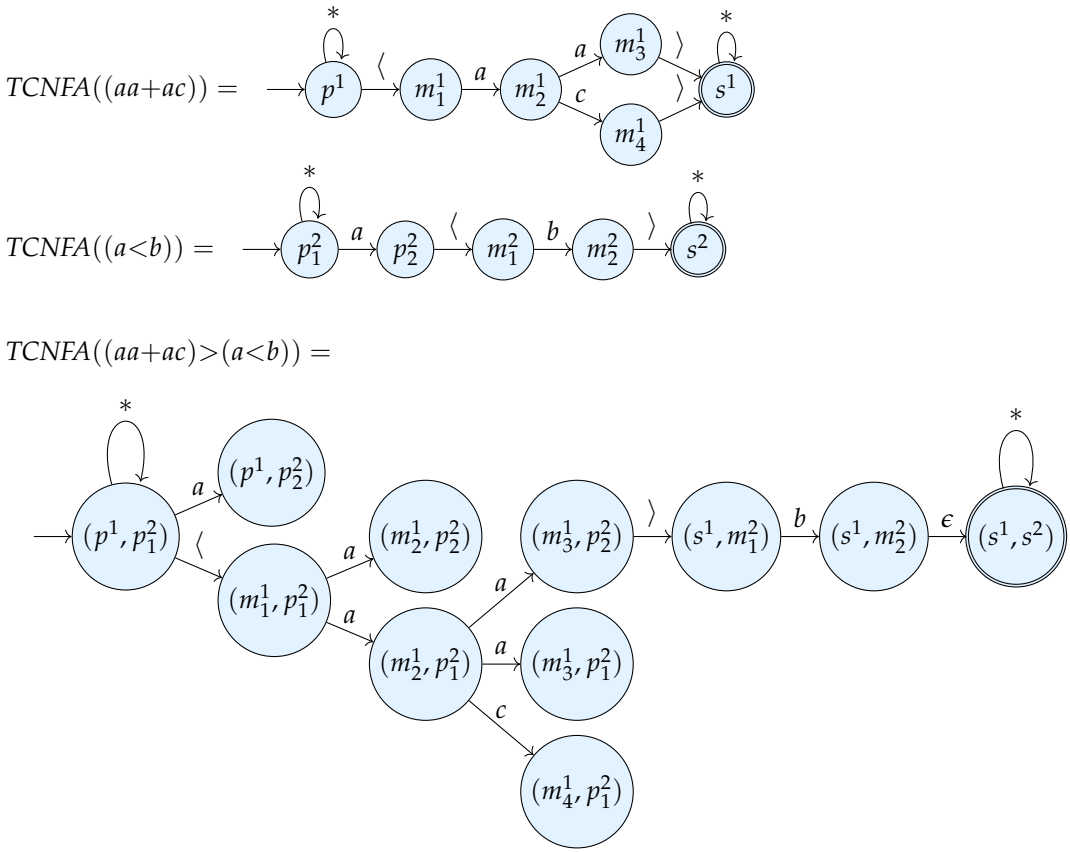

Figure 4: Lookahead TCNFA Example

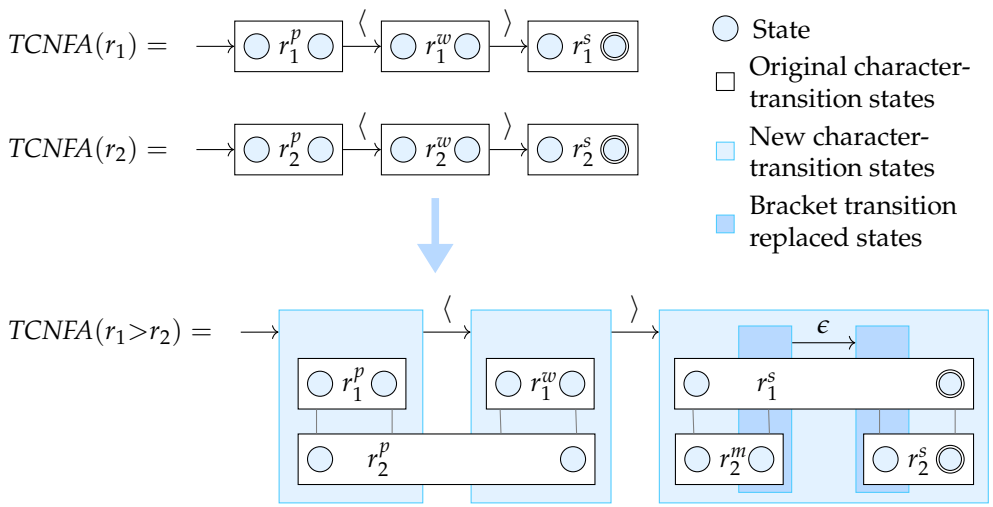

Figure 5: Lookahead TCNFA Structure

We can formally define $TCNFA(r_1{>}r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= Q_1^p \times Q_2^p \\
Q_w &= Q_1^w \times Q_2^p \\
Q_s &= (Q_1^s \times Q_2^w) \cup (Q_1^s \times Q_2^s) \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{\epsilon} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_2 \xrightarrow{\epsilon} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^p \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\langle} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\rangle} s_1' \in R_1 \wedge s_2 \xrightarrow{\langle} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in Q_1^s \wedge s_2 \in Q_2^w \wedge s_2 \xrightarrow{\rangle} s_2' \in R_2\} \\
i &= (i_1, i_2) \\
F &= F_1 \times F_2
\end{aligned}
$$

The definition for a lookbehind is symmetric. Defining a negative lookahead/lookbehind works similarly, except the complement of the TCNFA of r_2 is used. This is done by converting the TCNFA of r_2 to a TCDFA and setting its final states to $Q_2^s \backslash F_2$. Some special care has to be taken here regarding the match-end symbol of r_2, or else the described language will end up encoding something along the lines of $\{(p, w, s) \in \mathcal{L}(r_1) \mid \exists\ \alpha, \beta \in \Sigma^*\ .\ s = \alpha\beta \wedge (pw, \alpha, \beta) \notin \mathcal{L}(r_2)\}$, where the negation moved inside the existential quantifier.

The exact definitions for all constructions have been provided in Appendix A. This also includes the definition and explanation of the Kleene plus operator, which is the most complex of the operators. Finally we will give the definition of the tag operator here. We define $TCNFA((\langle t \rangle r_1)) = (Q_1^p, Q_1^w, Q_1^s, \Sigma, T, R, i_1, F_1)$ where:

$$
\begin{aligned}
R &= \{s \xrightarrow{\epsilon} s' \mid s \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s \xrightarrow{\epsilon} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{\langle} s' \mid s \in Q_1^p \wedge s \xrightarrow{\langle} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{\rangle} s' \mid s \in Q_1^w \wedge s \xrightarrow{\rangle} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{(a, T_1)} s' \mid s \in (Q_1^p \cup Q_1^s) \wedge s \xrightarrow{(a, T_1)} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{(a, \{t\} \cup T_1)} s' \mid s \in Q_1^w \wedge s \xrightarrow{(a, T_1)} s' \in R_1\}
\end{aligned}
$$

Appendix B contains more detail on the implementation of these TCNFA and the conversion algorithm. This includes condensed character transitions to deal with large alphabets and a normal form on these condensed TCNFAs for simple language equivalence testing.

We can now use these automaton to provide answers to questions about regular expressions in an automated way:

- Are two given regular expressions semantically equivalent? Answered by checking whether the TCNFAs of both expressions define the same language, using one of the following approaches:
  - Determine whether a bisimulation exists [14].
  - Determine whether the symmetric set difference of the languages is empty using union, complement, and product automata.
  - Check automata equality after normalization see Appendix B for more details [15].

- Do any of the words defined by a regular expression form a prefix of words of another regular expression? See overlap merging in Section 4.3.3.
- Does a given regular expression define empty words? Check whether the language of the product automaton with the $TCNFA(\epsilon)$ automaton is empty.

3.2 Conversion Grammars

The CG model is an extension of the formal CFG model. Because we are interested in encoding tokenizations into our grammars too, our grammars contain a set of tags used in the grammar as categories. Instead of using a sequence of terminal and non-terminal symbols for our productions, production components will be used. The universe of these production components is infinite, and is defined inductively in terms of the finite sets of terminal and non-terminal symbols:

- Any regular expression over the set of terminals is a production component
- Any reference to a non-terminal, combined with a sequence of tags is a production component
- Any production component with a positive/negative follow/precede restriction of another production component, is a production component
- Any production component with the language of another production component deleted, is a production component
- Any production component with the constraint that it has to appear at the start or end of a line, is a production component

In order to properly reason about the grammar transformations we will perform, we want to formalize these grammars together with the language and tokenizations they define. To do this, we will first look at traditional CFG formalizations. Traditional CFGs are defined by a tuple (V, Σ, R, s), where:

- V is a finite set of non-terminal symbols
- Σ is a finite set of terminal symbols
- $R \subset V \times (V \cup \Sigma)^*$ is a finite set of production rules
- $s \in V$ is the start symbol

Sentences in the language of such a grammar can be obtained by starting from the start symbol of the grammar, and applying substitutions rules defined in R until the resulting sequence only consists of terminal symbols. More precisely, we can define the language $\mathcal{L}(G)$ of a CFG $G = (V, \Sigma, R, s)$ as follows:

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid s \Rightarrow^* w\}$$

This $\Rightarrow^*$ relation used here is defined as the transitive reflexive closure of the relation $\Rightarrow \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ which is defined as follows:

$$\forall\, \alpha, \beta, \gamma \in (V \cup \Sigma)^*, S \in V \,.\, \alpha S \beta \Rightarrow \alpha\gamma\beta \quad \text{iff} \quad (S, \gamma) \in R$$

This way of defining the language of our CFG is simple and intuitive, as it is based on rewrites that can easily be performed by hand. But it will not work well for the production components of our language. The language of a CFG can also be defined in terms of fixed-points [16]. For every $v \in V$, let $D \subseteq (V \cup \Sigma)^*$ be the set of all sequences that v can substitute to: $\forall\, d \in D \,.\, (v, d) \in R$. Then we can create the a language equation for this non-terminal v as follows:

$$\mathcal{L}(v) = \cup_{d \in D}\, \mathcal{L}(d)$$

We then define the language of a symbol sequence $d \in (V \cup \Sigma)^*$ as follows:

$$\begin{aligned} \mathcal{L}(\epsilon) &= \{\epsilon\} \\ \mathcal{L}(a\alpha) &= \{a\} \ \cdot \ \mathcal{L}(\alpha) && \text{for } a \in \Sigma, \alpha \in (V \cup \Sigma)^* \\ \mathcal{L}(v\alpha) &= \mathcal{L}(v) \ \cdot \ \mathcal{L}(\alpha) && \text{for } v \in V, \alpha \in (V \cup \Sigma)^* \end{aligned}$$

Here the $\cdot$ operator represents pairwise concatenation between all sequences in each of the languages:

$$X \ \cdot \ Y = \{\alpha\beta \mid \alpha \in X, \beta \in Y\}$$

From all these language equations a single function f could be defined, which takes a vector of the current language for each symbol, and outputs the new language according to the equations. The language of a CFG $G = (V, \Sigma, R, s)$, is the language specified for s in the least fixed-point of f. For conciseness, we will say this is the least fixed-point of $L(s)$ from here on out. This language can be approximated by iteratively augmenting the language per symbol. The first approximation is initialized as an empty language for each non-terminal $v \in V$, namely: $\mathcal{L}_0(v) = \emptyset$. The next approximation can be calculated in terms of the previous: $\mathcal{L}_{i+1}(v) = \cup_{d \in D} \mathcal{L}_i(d)$, where $\mathcal{L}_i(d)$ uses the same equations as previously defined, except using approximation i of the language for each non-terminal. Since the target language is infinite in many cases, this approximation process would not terminate. Any sentence that is part of the language, does belong to approximation $\mathcal{L}_i(s)$ for some sufficiently large i however, making this approximation process valuable in understanding the actual infinite language.

Now we will define a similar formalization of our CG format, and use fixed-point equations to define the language of a given grammar. A CG is defined by a tuple (V, Σ, T, R, s), where:

- V is a finite set of non-terminal symbols
- Σ is a finite set of terminal symbols
- T is a finite set of tags
- $R \subset V \times \mathcal{C}(V, \Sigma, T)^*$ is a finite set of production rules
- $s \in V$ is the start symbol

We then define the production component universe $\mathcal{C}(V, \Sigma, T)$ inductively. We have two base-elements:

$$\begin{aligned} \mathit{ref}(v, t) &\in \mathcal{C}(V, \Sigma, T) && \text{for } v \in V, t \in T^* \\ \mathit{regex}(r) &\in \mathcal{C}(V, \Sigma, T) && \text{for } r \in \mathcal{R}(\Sigma, T^*) \end{aligned}$$

To assure that the language corresponding to this grammar is well defined, we have to add some additional constraints to the regular expressions that may be used:

- Positive lookarounds may not contain tags, in order to prevent tags from occurring in the prefix or suffix part of the regular expression's language
- Any nested tag declaration must be an extension of its closest ancestor tag. This makes sure a proper ordering is defined on the tags of the language. I.e. forall $(\langle t \rangle r)$ and their closest ancestor $(\langle t' \rangle r')$, we have that $\exists\ m \in T\ .\ t = t'm$.

In addition to these base-elements, the universe of productions components also contains several inductive

elements, all of which specify constraints:

$$\begin{aligned}
\mathit{follow}(s,f) &\in \mathcal{C}(V,\Sigma,T) && \text{for } s,f \in \mathcal{C}(V,\Sigma,T)\\
\mathit{not\text{-}follow}(s,f) &\in \mathcal{C}(V,\Sigma,T) && \text{for } s,f \in \mathcal{C}(V,\Sigma,T)\\
\mathit{precede}(s,p) &\in \mathcal{C}(V,\Sigma,T) && \text{for } s,p \in \mathcal{C}(V,\Sigma,T)\\
\mathit{not\text{-}precede}(s,p) &\in \mathcal{C}(V,\Sigma,T) && \text{for } s,p \in \mathcal{C}(V,\Sigma,T)\\
\mathit{except}(s,e) &\in \mathcal{C}(V,\Sigma,T) && \text{for } s,e \in \mathcal{C}(V,\Sigma,T)\\
\mathit{atStartOfLine}(s) &\in \mathcal{C}(V,\Sigma,T) && \text{for } s \in \mathcal{C}(V,\Sigma,T)\\
\mathit{atEndOfLine}(s) &\in \mathcal{C}(V,\Sigma,T) && \text{for } s \in \mathcal{C}(V,\Sigma,T)
\end{aligned}$$

The language of a CG is different than most typical languages. Similar to how the language universe for our custom regular expressions had to be augmented to support tags and lookarounds, we have to use a more complicated language universe for a CG in order to support precede and follow declarations and to encode tokenizations. It is not simply a subset of Σ^* as is the case with CFGs, but instead is a subset of $\Sigma^* \times (\Sigma \times T^*)^* \times \Sigma^*$. This means that every sentence in the language is a triple, consisting of:

- The prefix: a sequence of terminals allowed before the main body
- The main body: a sequence of terminal symbols, each accompanied by a sequence of tags that apply to that symbol
- The suffix: a sequence of terminals allowed after the main body

These tags represent the categories of our tokenization, and thus T^* represents a scope. Using the language $\mathcal{L}(G)$ of a conversion grammar G, we could define all possible tokenizations $\mathcal{T}(w,G)$ for a given input $w \in \Sigma^*$ as follows:

$$\mathcal{T}(w,G) = \{h \mid (\epsilon,(w,h),\epsilon) \in \mathcal{L}(G)\}$$

From our definition of $\mathcal{T}(w,G)$, it becomes clear that not every input sentence w necessarily has one unique tokenization. It may not have not have any tokenizations, or might instead have multiple valid tokenizations. If a sentence does not have any tokenization, we say it does not belong to the language. Syntax highlighters deal with the problem of an input possibly not having a tokenization by assigning default tokens to certain characters in the input while skipping them. This ensures that every possible input gets at least some tokenization, even if it is not a meaningful one. When there exists a language with multiple tokenizations, we say the grammar is tag-ambiguous. More formally, a grammar G is tag-ambiguous if and only if $\exists\ w \in \Sigma^*\ .\ |\mathcal{T}(w,G)| \geq 2$. Syntax highlighters always obtain a single tokenization per word by making greedy choices. These greedy choices are performed throughout the algorithm, which may lead to skipping of characters even if the given input had a properly defined tokenization if other choices were made earlier on. We will get back to this in Section 4.4.

We can now use fixed-point equations to define the language for a given grammar. For every $v \in V$, let $D \subseteq \mathcal{C}(V,\Sigma,T)^*$ be the set of all sequences that v can substitute to: $\forall\ d \in D\ .\ (v,d) \in R$. Then we can create the language equation for this non-terminal v as follows:

$$\mathcal{L}(v) = \cup_{d \in D}\ \mathcal{L}(d)$$

We then define the language of a sentential form $d \in \mathcal{C}(V,\Sigma,T)^*$ as follows:

$$\begin{aligned}
\mathcal{L}(\epsilon) &= \{\epsilon\}\\
\mathcal{L}(c\alpha) &= \mathcal{L}(c)\ \cdot\ \mathcal{L}(\alpha) && \text{for } c \in \mathcal{C}(V,\Sigma,T), \alpha \in \mathcal{C}(V,\Sigma,T)^*
\end{aligned}$$

To support this definition, we need to define the language for a single component. We also use a different definition for the concatenation operator $\cdot$ in order to consider the allowed prefixes and suffixes. This concatenation operation takes inspiration from our regular expression definition, and ensures that it only concatenates a triplet x with another triplet y, if the start of the suffix of x matches the body of y, and

symmetrically the end of the prefix of y matches the body of x. The binary concatenation operator $\cdot$ on two sets of triples is then more precisely defined as follows:

$$X \cdot Y = \{(p, (w_1 w_2, t_1 t_2), s) \mid (p, (w_1, t_1), w_2 s) \in X, (p w_1, (w_2, t_2), s) \in Y\}$$

Finally, we define the language for each of the different production component types:

$$\begin{aligned}
\mathcal{L}(\mathit{ref}(v,t)) &= \{(p', (w, tt'), s') \mid (p', (w, t'), s') \in \mathcal{L}(v)\} \\
\mathcal{L}(\mathit{regex}(r)) &= \{(p, (w, \mathcal{F}(T^w \cup \{\epsilon\})), s) \mid ((p, T^p), (w, T^w), (s, T^s)) \in \mathcal{L}(r)\} \\
\mathcal{L}(\mathit{follow}(s,f)) &= \{(p', (w,t), s') \in \mathcal{L}(s) \mid \exists\, \alpha, \beta \in \Sigma^*, \gamma \in T^* \,.\, \alpha\beta = s' \wedge (p'w, (\alpha, \gamma), \beta) \in \mathcal{L}(f)\} \\
\mathcal{L}(\mathit{not\text{-}follow}(s,f)) &= \{(p', (w,t), s') \in \mathcal{L}(s) \mid \neg\exists\, \alpha, \beta \in \Sigma^*, \gamma \in T^* \,.\, \alpha\beta = s' \wedge (p'w, (\alpha, \gamma), \beta) \in \mathcal{L}(f)\} \\
\mathcal{L}(\mathit{precede}(s,f)) &= \{(p', (w,t), s') \in \mathcal{L}(s) \mid \exists\, \alpha, \beta \in \Sigma^*, \gamma \in T^* \,.\, \alpha\beta = p' \wedge (\alpha, (\beta, \gamma), ws') \in \mathcal{L}(p)\} \\
\mathcal{L}(\mathit{not\text{-}precede}(s,f)) &= \{(p', (w,t), s') \in \mathcal{L}(s) \mid \neg\exists\, \alpha, \beta \in \Sigma^*, \gamma \in T^* \,.\, \alpha\beta = p' \wedge (\alpha, (\beta, \gamma), ws') \in \mathcal{L}(p)\} \\
\mathcal{L}(\mathit{except}(s,e)) &= \{(p', (w,t), s') \in \mathcal{L}(s) \mid \neg\exists\, \gamma \in T^* \,.\, (p', (w, \gamma), s') \in \mathcal{L}(e)\} \\
\mathcal{L}(\mathit{atEndOfLine}(s)) &= \{(p', (w,t), s') \in \mathcal{L}(s) \mid s' = \epsilon \vee \exists\, \alpha \in \Sigma^* \,.\, s' = \backslash\mathbf{n}\alpha\} \\
\mathcal{L}(\mathit{atStartOfLine}(s)) &= \{(p', (w,t), s') \in \mathcal{L}(s) \mid p' = \epsilon \vee \exists\, \alpha \in \Sigma^* \,.\, p' = \alpha\backslash\mathbf{n}\}
\end{aligned}$$

The language for a given regular expression relies on the tag-filter function $\mathcal{F}(S) \subseteq \mathcal{P}(T^*) \times T^*$, which is defined as:

$$\mathcal{F}(S) = t \in S \text{ such that } \neg\exists\, t' \in T \,.\, |t'| > |t|$$

For any regular expression with the required constraints, exactly one longest scope will exist.

The ref component simply augments the language of the non-terminal that is referenced by the specified tag-sequence. The variants of the follow and precede conditions use either the suffix or prefix of triples in the language of symbol s to filter out some entries. For example, the follow component only keeps the triples for which the suffix starts with a terminal symbol sequence that is part of the language of the second symbol f. Except filters out entries that are also contained in the second language, modulo the provided scopes. The end of line component makes sure that the suffix is either empty (indicating the end of the sentence) or indicates the start of a new line. Symmetrically the start of line components makes sure the prefix ends in a linefeed character. This formalization of the language of a grammar also highlights how all inductively defined components merely serve as additional constraints to filter certain phrases out of the language. This mirrors the intended behavior of analogous constructs in Rascal that are meant to reduce ambiguity.

The language $\mathcal{L}(G)$ of a given grammar $G = (V, \Sigma, T, R, s)$ is now defined as the least fixed-point of $\mathcal{L}(s)$. Note that the function representing our language equations is not monotonic, and thus there might not be unique least fixed-point. In fact, we can easily construct a grammar for which there is no least fixed-point, using a single production and start-symbol A where $A \rightarrow \mathit{except}(\mathit{regex}(a), \mathit{ref}(A, \epsilon))$. Using our iterative approximation scheme, we find the first approximation of the language includes a, then this phrase is filtered out in the second approximation, and reinserted in the third. More precisely, we find that any odd indexed approximation includes the phrase a, while any even indexed approximation does not. In order to prevent such behavior, CGs disallow recursive reference loops where the non-terminal reference appears as the second argument of a production component, which is used for filtering of entries. The resulting function is still not monotonic, but we believe it does have a well defined unique fixed-point, as it is no longer possible for the language of a given non-terminal shrinking to cause it to grow again later.

3.3 Grammar Relationships

In order for the CG format to be of use, it has to be related to the grammar formats we are trying to convert between: Rascal's format, and TextMate's format.

The CG format is designed in a way to make it easy to convert a Rascal grammar to a CG. The CG does not have nearly as many features as Rascal's format does, but any grammar making use of these features can be refactored into a grammar with an equivalent language that uses only the features we support. As terminal symbols, Rascal supports character classes, literal case sensitive strings, and literal case-insensitive

strings. All of these constructs can be mapped to regular expressions. Our formal regular expression model does not include character classes, but our practical implementation does support these as specified in Appendix B. Rascal's layout statements for defining optional whitespace are automatically interwoven into productions of the defined grammar, meaning that this feature is directly and automatically factored out. In order to make grammars more concise and require fewer non-terminals, Rascal supports syntax for regular operations. Rascal already provides a way of factoring out these operations, by considering them to be non-terminals and generating corresponding definitions for these non-terminals. Finally precedence and associativity can be specified for productions in Rascal to deal with ambiguity stemming from productions that are left or right-recursive. Such productions are common in grammars that describe binary operators, or unary prefix or suffix operators. Precedence and associativity however only deal with ambiguity regarding nesting of productions in parse trees, without affecting the language described by the grammar. Therefore, these constructs can simply be ignored in our mapping, without changing the intended language. All remaining features map onto features included in the CG format quite trivially. The CG format features production component constraints for each of Rascal's constraint types including precede, follow, reservation, and end of line constraints. Categories are attached as production attributes in Rascal, and can be transferred to the *ref* and *regex* components of a CG constructed for the corresponding production.

Lexing grammars are much more restrictive than our format, hence not every CG can be converted to a LG. For a CG to be converted to a LG, each non-terminal has to adhere to two requirements. It has to have an empty production, making every non-terminal optional, and every production must have one of the following shapes:

- $A \rightarrow \epsilon$
- $A \rightarrow \mathit{regex}(x)\ \mathit{ref}(A, \epsilon)$
- $A \rightarrow \mathit{regex}(x)\ \mathit{ref}(B, t)\ \mathit{regex}(y)\ \mathit{ref}(A, \epsilon)$

We will call a CG adhering to these constraints a LCG. This format forces the corresponding grammars to behave similar to how LGs behave. A non-terminal now essentially represents a pattern collection. After matching one of these patterns, we can continue matching other patterns of the same collection. This is enforced by requiring productions to be right-recursive. However, if all productions were right-recursive, the grammar would not contain any finite phrase. Therefore we enforce every non-terminal to have an empty production, simulating that every non-terminal can have its patterns match zero or more times. This empty production does not have to be mapped to any construct of a LG, and instead is implicitly present in the semantics of any LG. The second production form can be mapped to a simple pattern consisting of the regular expression x. Finally, the third production form can be mapped to a hierarchical pattern. Special care has to be taken regarding the accompanying non-terminal scope if it consists of more than 1 category, but we will cover this in more detail in the Chapter 4.5.1.

In the next chapter we will have a look at how this mapping is performed exactly, and at how we obtain a grammar with these constraints, from an arbitrary CG. To help with readability, we will from now on abstract away from the exact notation and formalism described in this chapter. These formalisms have been introduced to form a strong foundation that allows us to reason about language transformations, but are not very concise or well suited for giving an overview of the transformation pipeline. Instead we will make use of the following notational conventions:

- A: Uppercase characters at the start of the alphabet represent non-terminal symbols, as well as non-terminal symbol references when occurring as a production component: $A = \mathit{ref}(A, t)$ for arbitrary t
- $(\langle t \rangle A)$: Regex tag notation is also used to represent tag non-terminal references with specific tags, *e.g.* $(\langle t \rangle A) = \mathit{ref}(A, t)$
- A_i: A non-terminal symbol indexed at some i represents the i^{th} production of this non-terminal, *e.g.* given $A \rightarrow \alpha$, $A_1 = \alpha$
- $/a/$: Regular expressions are often wrapped in forward slashes, to distinguish them from other operations

- x: Lowercase characters at the end of the alphabet represent arbitrary regular expressions or regular expression components, *e.g.* $x = /a{>}b/$ or $x = regex(/a{>}b/)$
- $\dot{x}$: A regular expression character, or non-terminal reference character, with a dot specifies that the regular expression or reference does not specify any non-empty scopes. *e.g.* $\dot{x} = /a{>}b/$ or $\dot{A} = ref(A, \epsilon)$ but $\dot{x} \neq /(\langle t \rangle a{>}b)/$ and $\dot{A} \neq ref(A, t)$
- $\underline{x}$: A regular expression character with an underlines specifies that the regular expression does not contain the newline character
- α: Lowercase characters at the start of the greek alphabet represent arbitrary production component sequences, *e.g.* $\alpha = precede(ref(A, t), ref(B, \epsilon))\ regex(/a/)$. These are sometimes also used to represent other types of sequences.
- $[\![x]\!]$: Any production component nested in double brackets represents that the component is nested in zero or more inductive production components, *e.g.* $[\![A]\!] = precede(ref(A, t), ref(B, \epsilon))$

4 Conversion

In this section, a conversion pipeline is introduced. This pipeline ensures that any conversion grammar G_s is converted to a final LG. Before reaching the final LG, the transformation pipeline obtains a LCG G_l. This is expressed by the following conjecture:

Conjecture 1. The introduced pipeline transforms any CG G_s to a LCG G_l.

Proving termination of the pipeline forms the main difficulty in proving this conjecture, as discussed in Section 4.3. The pipeline attempts to ensure that the tokenizations of G_s for a given word w are also present in the tokenizations of G_l for w more formally phrased as: $\forall\ w \in \Sigma^*\ .\ \mathcal{T}(w, G_s) \subseteq \mathcal{T}(w, G_l)$. This is not a property that this pipeline can always ensure. In many cases we can however determine that the performed transformations are safe and retain this property. The pipeline will therefore keep track of a set of warnings and errors regarding tokenization equivalence. If no errors are generated, the lexing grammar should have an equivalent tokenization for all possible inputs. If errors are generated, it is possible that some inputs have different tokenizations. This information can be used to manually change G_s to obtain the desired G_l. This is expressed by the following conjecture:

Conjecture 2. If the conditions of Lemmas 1-3 are met during conversion from CG G_s to LCG G_l then G_l contains all tokenizations of G_s.

The mapping from our formal LCG to an actual LG can also be problematic. The semantics of LG are based on greedy choices, while LCG are not. Moreover, G_l may even be ambiguous, even if G_s was not. Therefore in order to ensure that the mapping is correct, the LCG should not contain any choices. What this entails exactly, will be discussed in Chapter 4.4. Before mapping G_l to an industry format, we will map it to an intermediate custom format, in order to abstract away from exact syntax. We introduce Scope grammars and PDA grammars to represent these industry formats for which exact semantics will be specified. These semantics obtain exactly a single tokenization for any possible word over the grammar's alphabet. Finally, a LCG should not have any left-recursive cycles either. More formally, for any non-terminal symbol A that can be substituted by $\alpha A \beta$ using any number of substitutions, α should contain a regular expression that does not accept the empty sequence ϵ. If this property does not hold, a tokenizer might get stuck in a non-productive loop and not halt. Using some special care, we can ensure that there are no non-productive recursive loops in grammars obtained from our pipeline. These conditions together form our last conjecture:

Conjecture 3. If the conditions of Lemmas 4-6 are met when obtaining a Scope or PDA grammar G_f from LCG G_l then for any word tokenized by G_l, G_f obtains a single equivalent tokenization.

Note that this conjecture states that the single tokenization of G_f for a given word is equivalent to the tokenizations obtained from G_l, which implies G_l contains either no tokenization or a single tokenization per word. This is specified by the condition of Lemma 4.

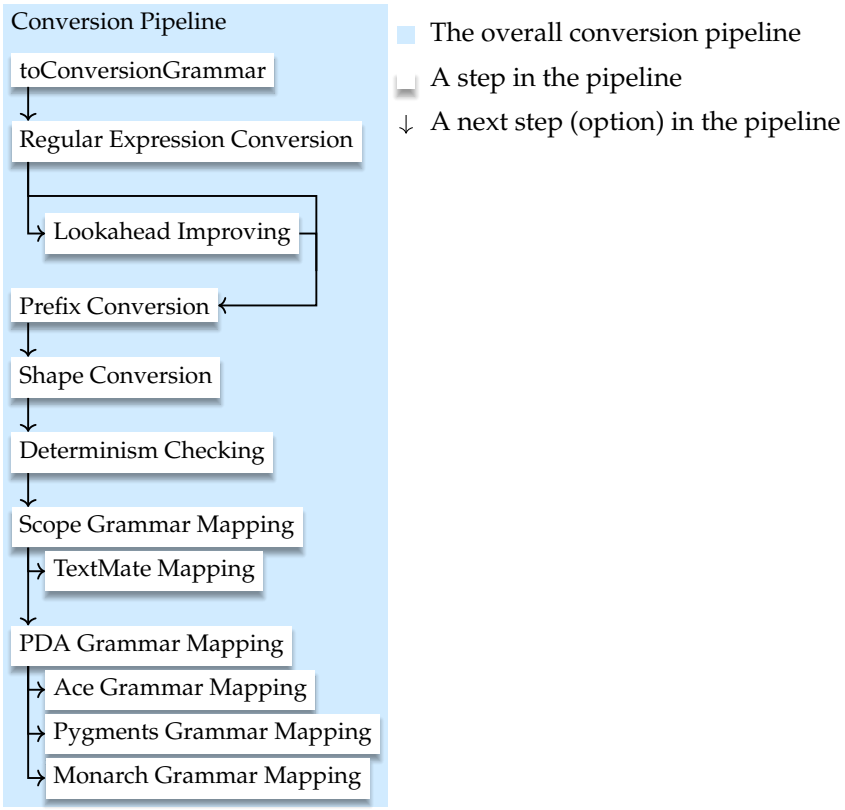

Figure 6: Conversion Pipeline Overview

We will first describe steps to obtain a LCG: regular expression conversion, prefix conversion, and shape conversion. We then specify checks that can establish whether the resulting grammar is free of choices. After this, the intermediate formal LGs Scope grammars and PDA grammars are introduced, and a mapping to industry formats is described. Finally, an optional step is introduced to deal with some of the potential non-determinism in output grammars by means of improving lookaheads in the input grammars. Figure 6 provides an overview of the complete pipeline.

4.1 Regex Conversion

The first step in the conversion pipeline is to convert elements in the grammar into regular expressions whenever possible. This is an important step, because LGs can deal with arbitrarily complex regular expressions, but put many restrictions on the production format. Every transformation made in this step ensures that the resulting grammar describes a fully equivalent language, contributing to the following lemma:

Lemma 1. *The regular expression conversion step preserves all the grammar's tokenizations without adding new tokenizations.*

After this, all remaining constraint production components are removed from the grammar. The goal is to have pushed all constraints into the level of regular expressions, but in case this was not successful, a warning is generated. These warnings do not relate to our tokenization equivalence assurance, but rather try to help pinpoint what might have caused errors generated later in the pipeline. Removing constraints will only broaden the language, and not remove any tokenizations. Hence if the final grammar is free of choices and has no errors, its tokenization is maintained despite these warnings.

When converting CFG constructs into regular expression constructs, there are two limiting factors present in tokenizers that we have to consider: regular expression capture groups only capture the last occurrence, and tokenizers operate line by line. This former behavior prevent us from using tag constructs within iteration constructs, since this would result in tokenizers only properly assigning tokens to the last occurrence in the match. The latter behavior prevents us from concatenating anything after a newline character in a regular expression. If we were to create a regular expression defining a language in which a linefeed is followed by any other character, the tokenizer would not be able to match this entry at all.

The conversion is based on a set of rewrite rules on productions of a conversion grammar. Inference rule notation is used for these, to clearly distinguish the rewrite rules in the grammar itself from the rewrite rules performed on the grammar. This notation represents that the premises can be replaced by the given conclusion, while preserving our desired tokenization property. The notation $A \not\to$ is used to indicate that symbol A has no productions other than the ones listed as premises. Below is an example for a grammar rewrite rule specifying that a sub-sequence of a production can be replaced by a non-terminal with a production of that sub-sequence:

$$\frac{A \to \alpha\ \beta \quad B \not\to}{A \to B\ \beta \quad B \to \alpha}\ \textit{naming}$$

$$\frac{A \to \alpha\ \underline{x}\ y\ \beta}{A \to \alpha\ /\underline{x}y/\ \beta}\ \textit{concatenation}$$

$$\frac{A \to \alpha\ x\ \beta \quad A \to \alpha\ y\ \beta}{A \to \alpha\ /x{+}y/\ \beta}\ \textit{alternation}$$

$$\frac{A \to \alpha\ x\ \beta \quad A \to \alpha\ \beta}{A \to \alpha\ /x{+}\epsilon/\ \beta}\ \epsilon\textit{-alternation}$$

$$\frac{A \to \alpha\ [\![(\langle s\rangle B)]\!]\ \beta \quad B \to x \quad B \not\to}{A \to \alpha\ [\![/(\langle s\rangle x)/]\!]\ \beta}\ \textit{substitution}$$

$$\frac{A \to \alpha\ (\langle s\rangle B)\ \beta \quad B \to [\![x]\!]\ \cdots\ [\![(\langle t\rangle A)]\!] \quad B \not\to}{A \to \alpha\ [\![/(\langle s\rangle x)/]\!]\ \cdots\ [\![(\langle st\rangle A)]\!]\ \beta}\ \textit{sequence-substitution}$$

$$\frac{A \to /(\langle s\rangle \underline{\dot{x}})/\ \dot{A} \quad A \to \alpha \quad A \not\to}{A \to /(\langle s\rangle \underline{\dot{x}}^*)/\ \alpha}\ \textit{right-repetition}$$

$$\frac{A \to \alpha \quad A \to \dot{A}\ /(\langle s\rangle \underline{\dot{x}})/ \quad A \not\to}{A \to \alpha\ /(\langle s\rangle \underline{\dot{x}}^*)/}\ \textit{left-repetition}$$

$$\frac{A \to /(\langle s\rangle \underline{\dot{x}})/\ \dot{A} \quad A \to \alpha \quad A \to \dot{A}\ /(\langle s\rangle \underline{\dot{x}})/ \quad A \not\to}{A \to /(\langle s\rangle \underline{\dot{x}}^*)/\ \alpha\ /(\langle s\rangle \underline{\dot{y}}^*)/}\ \textit{bi-repetition}$$

$$\frac{A \to \alpha\ \textit{except}(x, y)\ \beta}{A \to \alpha\ /x{-}y/\ \beta}\ \textit{delete-lowering}$$

$$\frac{A \to \alpha\ \textit{follow}(\underline{x}, y)\ \beta}{A \to \alpha\ /\underline{x}{>}\dot{y}/\ \beta}\ \textit{follow-lowering}$$

$$\frac{A \to \alpha\ \textit{not-follow}(\underline{x}, y)\ \beta}{A \to \alpha\ /\underline{x}{\not>}y/\ \beta}\ \textit{not-follow-lowering}$$

$$\frac{A \to \alpha\ \textit{precede}(x, \underline{y})\ \beta}{A \to \alpha\ /\underline{\dot{y}}{<}x/\ \beta}\ \textit{precede-lowering}$$

$$\frac{A \to \alpha\ \textit{not-precede}(x, \underline{y})\ \beta}{A \to \alpha\ /\underline{y}{\not<}x/\ \beta}\ \textit{not-precede-lowering}$$

$$\frac{A \to \alpha\ \textit{atEndOfLine}(\underline{x})\ \beta}{A \to \alpha\ /\underline{x}((\not>.){+}(>\backslash\mathbf{n}))/\ \beta}\ \textit{eol-lowering}$$

$$\frac{A \to \alpha\ \textit{atStartOfLine}(x)\ \beta}{A \to \alpha\ /((.\not<){+}(\backslash\mathbf{n}<))x/\ \beta}\ \textit{sol-lowering}$$

Rewrite 1: Regular Expression Grammar Rewrite Rules

The regex conversion is based on repeated application of the Rewrite Rules 1. For either of the substitution rules, special care has to be taken when adding a tag to an existing regular expression, in order to ensure scope order is properly encoded. If a scope $s \in T^*$ is added to a regular expression x, resulting in regular expression $(\langle s\rangle x')$, x' should be a copy of x where s has been concatenated in front of any tag constructs present in x. Assuming that x itself follows our regex requirements of not containing tags in lookarounds, $(\langle s\rangle x')$ will also adhere to the requirements. The dot in the end-of-line and start-of-line lowering transformations represents an alternation of all symbols in the alphabet, meaning that it would match

any character.

In consideration of tokenizers operating on one line at a time, no newline character may be present in the regular expression that forms the start of a sequence. This can be seen in the rewrite rules for concatenation, repetition, and lookaround-lowering. The language of the expression resulting from the start-of-line lowering transformation does not adhere to this, but the corresponding language always includes this same phrase without the newline character prefix due to the provided alternative. Therefore tokenizers will deal with this expression appropriately, even if executed line by line. Similarly, no scopes are allowed within the iteration structures, which is enforced by the rewrite rules. The rewrite rules for precede and follow do apply to regular expressions regardless of the presence of scopes, but the tag structures have to be filtered out from the lookaround. For instance $\mathit{follow}((\langle s\rangle z), (\langle t\rangle w))$ will be rewritten to $/(\langle s\rangle z) > w/$. This preserves the semantics, because scopes in constraint components do not become part of the language.

In addition to these rules, we also apply tag lifting. This attempts to merge tag constructs together whenever possible. Consider the expression $/(\langle s\rangle \dot{x})(\langle s\rangle \dot{y})/$, this expression would not match the required structure for any of the repetition rewrites, while the expression $(\langle s\rangle \dot{x}\dot{y})$ with an equivalent language does. This process of merging tags constructs could be described in terms of rewrite rules, but can be performed more generally by analyzing a regular expressions' language using TCNFAs. Tag usage in transitions of a TCNFAs can easily be analyzed to determine whether it can be lifted. Any tag that appears in exactly all character transitions going out of main-states can safely be lifted. Lifting of a tag t from expression x is done by filtering out all tag constructs specifying t resulting in expression x', and then using $/(\langle t\rangle x')/$ as the lifted expression.

In the code implementing regular expressions, a special constructor is provided that pairs a regular expression with a TCNFA. This is used to cache the TCNFA for a corresponding regular expression, such that it does not have to be recalculated every time some language check has to be performed on a given regular expression. When applying any of these transformation rules, the corresponding TCNFA is also immediately calculated and cached for the new expression.

These rewrite rules are applied in a fixed order, to prevent unnecessary condition checking. Empty non-terminals are immediately removed from the language by applying the sequence-substitution rule specifically on the non-terminals specifying only an empty sequence, *e.g.* $B \to \epsilon$. Similarly, the concatenation rule is applied exhaustively on all productions immediately. After this a loop is entered that keeps repeating until no transformations are being made anymore. Within this loop, every non-terminal is iterated over, and the following compound transformations are attempted in order:

- Unioning: applies both alternation transformations in exhaustively on a non-terminal. If successful, this also exhaustively applies concatenation and scope lifting on the resulting productions
- Substitution: Attempts to apply the substitution transformation to remove this symbol from the grammar. If successful, this also exhaustively applies concatenation, constraint lowering, and scope lifting on the resulting productions
- Repetition: Attempt to apply any of the repetition transformation rules to obtain a single production. If successful, this also exhaustively applies concatenation and scope lifting on the resulting production

When none of these compound transformations are effective on any of the non-terminals anymore, sequence-concatenation is applied on all symbols and if effective, the main loop is entered again to perform more compound transformations. This repeats until sequence-concatenation can not be applied anymore. Sequence concatenation is performed at the very end, because it is more generally applicable than regular concatenation, but regular concatenation has priority. This process trivially terminates, since applying any of the compound rules will result in fewer non-terminal symbols in the grammar, or an equal number of non-terminal symbols and a smaller number of production components in the grammar.

After all rules have been applied exhaustively, the remaining constraints are removed from the grammar, and corresponding warnings are generated.

4.2 Prefix Conversion

After regex conversion finishes, we know that every production in the grammar only contains reference or regex production components and does not contain any hierarchical components anymore. The second step

of the conversion applies transformations to make every production start with a regular expression. This is done because LGs patterns always start with a regular expression used to deterministically choose the next pattern to apply. This conversion step ensures that every production in the resulting grammar starts with a regular expression, and that the language of the input grammar is a subset of the output grammar. As such, it ensures that the original tokenization is maintained, but it might introduce additional tokenizations for a given input, making the tokenization ambiguous. This superset guarantee can not always be achieved, in which case corresponding errors are generated. This is for instance the case when a left-recursive pattern specifies a scope to be applied to the left recursion. This conversion step also ensures that any cycles are removed.

$$\frac{A \to B\ \alpha \quad A \neq B}{A \to \alpha \quad A \to B_1\ A \quad \cdots \quad A \to B_n\ A}\ \textit{left-expansion}$$

$$\frac{A \to (\langle\epsilon\rangle A)\ \alpha}{A \to \alpha \quad A \to \epsilon \quad A \in S}\ \textit{detect-self-recursion}$$

$$\frac{A \to \alpha\ B \quad A \in S \quad A \neq B}{A \to \alpha\ B\ A}\ \textit{symbol-right-recursion}$$

$$\frac{A \to \alpha\ x \quad A \in S}{A \to \alpha\ x\ A}\ \textit{regex-right-recursion}$$

$$\frac{A \to /x{+}\epsilon/\ \alpha}{A \to x\ \alpha \quad A \to \alpha}\ \epsilon\textit{-cycle-removal}$$

Rewrite 2: Prefix Conversion Grammar Rewrite Rules

The step is based on the Rewrite Rules 2. Note that these rewrite rules do not ensure that the language of the resulting grammar is equivalent. Instead, they ensure that the language of the resulting grammar is a superset of the original language. This is obvious for the first rewrite rule, which expands the productions of an non-terminal B into A and specifies that anything else of A may now follow it, which now includes the remainder of the original production. The self-recursion and right-recursive rules are a bit more complex, and have to be considered in combination with one and another. Set S tracks what symbols of the grammar are left-recursive. If at any point a left-recursive self-loop is detected, this loop is removed and the symbol as a whole is said to be left-recursive. For this rewrite rule to indeed maintain the language, $A \in S$ should be seen as a guarantee that any right-hand side of a production in A ends with a A, except for the empty production. This is achieved if the right-recursion rules are exhaustively applied. Consider non-terminal A with left-recursive production $A \to A\ \alpha$ and base production $A \to x\ \beta$. For any phrase in the language, the left-recursion of $A\ \alpha$ must eventually terminate, so at some point this A is substituted for $x\ \beta$, *e.g.* $A \to A\ \alpha \to A\ \alpha\ \alpha \to x\ \beta\ \alpha\ \alpha$. Now if instead all productions are made to be right recursive, resulting in $A \to \alpha\ A \quad A \to x\ \beta\ A \quad A \to \epsilon$, the same final phrase is obtainable as follows: $A \to x\ \beta\ A \to x\ \beta\ \alpha\ A \to x\ \beta\ \alpha\ \alpha\ A \to x\ \beta\ \alpha\ \alpha$. Clearly this also allows for derivations that were not part of the original language, but all that matters is that it guarantees the original phrases can also still be derived. Finally the ϵ-removal transform also maintains the exact language by simply splitting the given production in two, eliminating the epsilon match in the regular expression. There are however regular expressions that do not match this exact pattern, and also accept empty matches, such as $/(a{+}\epsilon)(b{+}\epsilon)/$. Therefore TCNFAs are used to detect presence of empty phrases in the language of the regular expression x, in which case x is said to be nullable, and an inductive function is used to obtain a regular expression y whose language is equivalent to x's except for any empty phrases being removed.

In order to define such a function, we generalize our goal. Given a regular expression x, we want to obtain three regular expressions. More precisely we specify a function $D : \mathcal{R}(\Sigma, T) \to \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T)$ which given a regular expression x outputs three regular expression m, e, and c such that:

- $\mathcal{L}(x) = \mathcal{L}(m{+}e{+}c)$

- $e = \epsilon \ \vee \ e = \mathbf{0}$
- $\mathcal{L}(\epsilon) \cap \mathcal{L}(m) = \emptyset$, *i.e.* m does not contain any empty phrases
- $\mathcal{L}(c) \backslash \mathcal{L}(\epsilon) = \emptyset$, *i.e.* c only contains empty phrases

Additionally, we would like to guarantee that $\mathcal{L}(\epsilon) \neq \mathcal{L}(c)$, since c should represent empty matches with additional constraints such as $/\epsilon{>}a/$. Being able to fully accurately separate empty-match regular expressions with restrictions from those without restrictions has practical uses, but is not strictly required for our algorithm, and hence won't be covered here due to the extra complexity. To achieve the desired specification, an inductive function can be used. Let r_1 and r_2 be regular expressions, and let $(m_1, e_1, c_1) = D(r_1)$ and $(m_2, e_2, c_2) = D(r_2)$ represent the inductive results. Then the function D can be defined as follows:

$$
\begin{aligned}
D(\mathbf{0}) &= (\mathbf{0},\ \mathbf{0},\ \mathbf{0}) \\
D(\mathbf{1}) &= (.^{+},\ \epsilon,\ \mathbf{0}) \text{ where } . = +_{a \in \Sigma}\ a \\
D(\epsilon) &= (\mathbf{0},\ \epsilon,\ \mathbf{0}) \\
D(a) &= (a,\ \mathbf{0},\ \mathbf{0}) \\
D(r_1{+}r_2) &= (m_1{+}m_2\ e_1{+}e_2,\ c_1{+}c_2) \\
D(r_1 r_2) &= (m_1 m_2{+}m_1 e_2{+}m_1 c_2{+}e_1 m_2{+}c_1 m_2,\ e_1 e_2,\ e_1 c_2{+}e_1 e_2{+}c_1 e_2) \\
D(r_1^{+}) &= (r_1^{*} m_1 r_1^{*},\ e_1,\ c_1^{+}) \\
D(r_1{>}r_2) &= (m_1{>}r_2,\ \mathbf{0},\ (e_1{+}c_1){>}r_2) \\
D(r_1{<}r_2) &= (r_1{<}m_2,\ \mathbf{0},\ r_1{<}(e_2{+}c_2)) \\
D(r_1{\not>}r_2) &= (m_1{\not>}r_2,\ \mathbf{0},\ (e_1{+}c_1){\not>}r_2) \\
D(r_1{\not<}r_2) &= (r_1{\not<}m_2,\ \mathbf{0},\ r_1{\not<}(e_2{+}c_2)) \\
D(r_1{-}r_2) &= (m_1{-}r_2,\ \mathbf{0},\ (e_1{+}c_1){-}r_2) \\
D((\langle t \rangle r_1)) &= ((\langle t \rangle m_1),\ e_1,\ c_1) \\
D((r_1)) &= (m_1, e_1, c_1)
\end{aligned}
$$

In this definition, the alphabet must be known to perform the transformation of $\mathbf{1}$, since an alternation of all possible symbols in the alphabet is used here. Alternatively we could simply use $\mathbf{1}{-}\epsilon$, but we want to try minimize usage of the subtraction operator. We do this because most regular expression engines do not support subtraction, but do have concise notation to represent the entire alphabet. The iteration case in this function is particularly interesting. One might expect that the language r_1^{+} cannot contain any empty strings, because it requires at least one iteration of r_1. This is however clearly not the case, since the language of r_1 might include an empty phrase itself, meaning that one iteration of an empty phrase match still results in an empty phrase. Then one might expect that m_1^{+} would be sufficient, since it forces at least one iteration to be present, and this iteration to be non-empty. We have to be a bit more careful however, we want to describe the original expression, but force at least one iteration to be non-empty, which more directly translate to the following: $r_1^{*} m_1 r_1^{*}$. These expressions happen to describe an identical language for regular expressions without tags, since any iterations that do not match any characters can at most restrict what may be seen before or after, and thus do not augment the language compared to only taking non-empty iterations. This is however not the case when tags are present in positive lookarounds, since they do not only restrict the language, but also modify the characters matched before or after this iteration. Similarly the iteration is important to be included in the empty constraints part of this regular expression: c_1^{+}. Only performing a single iteration is not equivalent here, if multiple expression can assign different tags. Additional simplification rules can be used for this inductive function, to filter unnecessary alternations and other constructs. Consider for instance the expressions used for concatenation, if $c_1 = \mathbf{0}$ then $\mathcal{L}(c_1 m_2) = \mathcal{L}(\mathbf{0})$, and $\mathcal{L}(\mathbf{0}{+}x) = \mathcal{L}(x)$, hence the term $c_1 m_2$ could be stripped from the alternation expressions.

The specified rewrite rules are applied per symbol on the entire grammar iteratively until no more rewrites can be performed. We keep track of the set of all left-recursive symbols, which is initialized to be

empty. Then per symbol, we obtain a list of new productions by mapping each production to a new one with transformation rules applied. We also always add the empty production $A \rightarrow \epsilon$, since we know our final LCG will have to include this in any case. The first step in the mapping of a production is checking whether it is of the shape $A \rightarrow x\ \alpha$ and x is nullable. If this is the case, D is used to extract a non-nullable expression y, and the production is mapped to $A \rightarrow y\ \alpha$ while the remainder $A \rightarrow \alpha$ is still processed for further mapping. α could start with another nullable regular expression, so this first step is performed iteratively. Then we check whether the production is of the shape $A \rightarrow (\langle t \rangle B)\ \alpha$, where we distinguish two cases $B \neq A$ and $B = A$. If the former is the case, the production is mapped to $A \rightarrow \alpha$ and every production of $B \rightarrow \beta$ is copied as $A \rightarrow \beta\ A$. We call this symbol expansion. During this mapping process, special care is taken regarding scopes. If $t \neq \epsilon$, this scope t has to be transferred to the expanded productions: $A \rightarrow (\langle t \rangle \beta)\ A$. There is no scoping construct for arbitrary sequences β in our conversion grammar format, so these scopes will have to be carried to the individual components in the sequence, similar to what we saw in the sequence substitution used in regular expression conversion. In the later case where $B = A$, we simply map the production to $A \rightarrow \alpha$ and add A to the set of left-recursive symbols. If $t \neq \epsilon$ we generate an error, since this scope can not be properly dealt with. In this case, we use Rewrite Rule 3, and tokenization is not guaranteed. More constructively we can say:

Lemma 2. *If for all encountered productions matching $A \rightarrow (\langle t \rangle A)\ \alpha$ for some sequence α we have $t = \epsilon$, the grammar's tokenizations are preserved by the prefix conversion step.*

This lemma follows from the set of transformations being language preserving when performed exhaustively. A proof for this should be worked out in further detail in the future.

Finally, if A belongs to the left-recursive symbols set, we check if the production $A \rightarrow \alpha$ matches $\alpha = \beta\ (\langle \epsilon \rangle A)$ for some β and if not, is transformed to $A \rightarrow \alpha\ (\langle \epsilon \rangle A)$. By performing these transformations iteratively until no more transformations apply, the right-recursion property will eventually be reached, even if it is not reached immediately when left-recursion is detected. It is not obvious that this procedure eventually terminates. In case the grammar contains recursive references, you might think that we could keep performing symbol expansion forever. This is however not the case, because every expansion will reduce the length of the recursive loop by one, until eventually a direct self-recursion is reached and dealt with properly. Hence this conversion step will eventually terminate, and ensure all productions start with regular expressions.

$$\frac{A \rightarrow (\langle t \rangle A)\ \alpha \quad t \neq \epsilon}{A \rightarrow \alpha \quad A \rightarrow \epsilon \quad A \in S \quad \mathit{inapplicableScopeError}}\ \mathit{detect\text{-}self\text{-}recursion\text{-}error}$$

Rewrite 3: Prefix Conversion Grammar Unsafe Rewrite Rule

4.3 Shape Conversion

After prefix conversion is done, every reachable symbol contains an empty production, every reachable production only consists of reference and regular expression components, and every reachable production starts with a regular expression. The next step is to convert every production to the exact shape we need, meaning that it should be of the form $A \rightarrow x\ B\ y\ (\langle \epsilon \rangle A)$, $A \rightarrow x\ (\langle \epsilon \rangle A)$, or $A \rightarrow \epsilon$. Additionally we would like to deal with the issue of non-determinism here, making sure that only a single production is applicable at a time. Similar to the prefix conversion step, our goal is to make sure that the language of the output grammar includes the original language, not for the languages to be equivalent. This conversion does not properly deal with all possible scenarios, and appropriate errors for these cases will be generated. Such errors are only generated for scopes on non-terminal symbols, or left-recursive loops.

This step provides the following guarantee about tokenization preservation:

Lemma 3. *If none of the premises of Rewrite Rules 5, 8, 12, or 18 apply during conversion, the grammar's tokenizations are preserved by the shape conversion step*

This lemma follows from each transformation rule being language preserving, and the initial productions of $convSeq(p)$, $unionRec(O)$, and $closed(s, c)$ realizing the language that is expected from the symbol.

The shape conversion step does not merely convert a given grammar; instead, it constructs a new grammar based on the one provided. It does however collect all productions including the provided source productions in a single grammar and replaces the start symbol. This makes it so most original symbols become unreachable. This way the reachable symbols can be constructed to have the correct shape, while still operating on only a single grammar. After conversion has fully finished, all unreachable symbols will be removed from the grammar. We define three custom non-terminal symbol constructors that are used to construct these new productions:

- $convSeq(p)$ for $p \in (\mathcal{C}(V_i, \Sigma, T) \cup TCNFA(\Sigma, T^*))^*$
- $unionRec(O)$ for $O \subseteq V_i$
- $closed(s, c)$ for $s, c \in V_i$

Here V_i represents the non-terminal set of the CG that is operated on at some moment in time. When a new non-terminal is created using one of these constructors, this set of non-terminals is augmented and defines the new set of terminals V_{i+1}. The structure of these new non-terminals plays no role in the semantics of the CG that includes them, but it does play a role within the conversion algorithm. Each of these non-terminals also has a corresponding specification, which the productions we create for them try to achieve:

- $convSeq(p)$ defines the same language as its arguments: $\mathcal{L}(convSeq(p)) = \mathcal{L}(p)$. The symbol only contains a single production, starting with a regular expression.
- $unionRec(O)$ defines a superset of any sequence consisting of these non-terminals: $\mathcal{L}(unionRec(O)) \supseteq (\cup_{o \in O} \mathcal{L}(o))^*$. These symbols always contain an empty production. Every non-empty production starts with a regular expression, and no production contains two consecutive non-terminals.
- $closed(s, c)$ defines a superset of sequences with the first and second argument alternating: $\mathcal{L}(closed(s, c)) \supseteq (\mathcal{L}(s) \cdot \mathcal{L}(c))^*$. These symbols always contain an empty productions, and every production is of one of the valid shapes for LCGs: $A \to x\ B\ y\ (\langle \epsilon \rangle A)$, $A \to x\ (\langle \epsilon \rangle A)$, or $A \to \epsilon$.

Additionally, we introduce the notion of alias productions and symbols, which simply redirect to another symbol. Thus a given symbol A is said to be an alias, if it only has a single production of the shape $A \to (\langle \epsilon \rangle B)$. The exact production requirements of the custom constructor symbols described may not be satisfied if the symbol is an alias, but these aliases will be inlined eventually.

The conversion starts by replacing the start symbol s by $closed(s, convSeq(E))$ where $E = /\epsilon \not> ./$ representing the end of file match. We know no symbols may follow E, and thus the specification states the resulting language is a superset of $(\mathcal{L}(s) \cdot \mathcal{L}(E))^* = \{\epsilon\} \cup \mathcal{L}(s) \cdot \mathcal{L}(E)$. For tokenization we care about inputs that have an empty prefix and suffix, thus this start symbol will correctly include our specified tokenization.

The remainder of the algorithm attempts to generate productions for all custom symbols reachable from the start symbol, including the stat-symbol itself. Creating the productions for one symbol may introduce references to new custom symbols that are not defined in the grammar yet. Therefore an iterative approach is used that continues until no new symbols are left to be defined:

1. Find all reachable $unionRec(O)$ symbols that have not yet been defined
2. If such symbols are found, define them and return to step 1
3. Otherwise deduplicate the grammar and continue
4. Find all reachable $closed(s, c)$ symbols that have not yet been defined
5. If such symbols are found, define them and return to step 1
6. Otherwise conversion is finished

The deduplication of the grammar is what introduces alias symbols. This attempts to detect different symbols that have equivalent languages, and introduces aliases for them. Additionally any occurrences of the symbol in the grammar is substituted by the symbol it is aliasing. It is important to keep the alias in the grammar however, to prevent the same symbol from being generated again in the future, when we know it will simply be replaced again. Using aliases like this makes the grammar smaller, but also may speed up conversion by preventing generation of entire trees of custom symbols, which ultimately would all have another symbol in the grammar with an equivalent language. Detecting equivalent symbols is done using partition refinement. We only consider symbols that already have defined productions here. A set of equivalence classes is defined, which is initialized to only contain one class consisting of all symbols in the language. Then we find a production in the grammar which partitions a given equivalence class into two non-empty classes: one that contains this production, and one that does not. If no such production can be found anymore, the current partition accurately describes equivalence classes. We will not fully define the type of equivalence used here, but it is clear that it ensures the languages of symbols in each class are equivalent. This does however not guarantee that symbols in different classes describe different languages, and thus this notion of equivalence should not be mistaken for language equivalence. Instead of testing for fully equivalent productions, we can once again replace regular expressions by TCNFAs in order to abstract over the exact used syntax. Moreover, when checking of two productions are equivalent, we do not require reference non-terminals to be equivalent, but only to belong to the same equivalence class of the current partition. Additionally two symbols $closed(s, c)$ and $closed(s', c)$ are determined to be equivalent if s and s' belong to the same equivalence class. This helps us to detect equivalences between productions that rely on symbols that have not yet been defined, which are assumed to belong to different classes by default.

4.3.1 Sequence Definition

The component sequence $convSeq(p)$ is used to allow a single symbol to refer to a given production component sequence. We construct such a symbol from a given production component sequence, and when we find that the symbol is already in the grammar, we do not have to redefine it. TCNFAs are used in the constructor instead of regular expressions in order to abstract over the exact syntax being used, to increase the chances of the sequence already being present in the grammar. These productions are generated at the time the non-terminal is constructed. In order to ensure these productions start with a regular expression, we use transformation rules on the symbol itself. These rules are similar to production rewrite rules, but operate on symbols directly before adding them to a production. Rewrite Rule 4 specifies this relaxation rule that is language preserving, while Rewrite Rule 5 generates an error when applied. The other two symbols can be created when a symbol according to the defined specification is needed, and their productions can later be generated based on the arguments of the constructors.

$$\frac{convSeq((\langle\epsilon\rangle A_1)\ \cdots\ (\langle\epsilon\rangle A_n)\ x\ \alpha)}{unionRec(\{A_1\ \cdots\ A_n,\ convSeq(x\alpha)\})}\ \textit{prefix-sequence}$$

Rewrite 4: Prefix Sequences

$$\frac{convSeq((\langle t_1\rangle A_1)\ \cdots\ (\langle t_n\rangle A_n)\ x\ \alpha)\quad t_1 \neq \epsilon \vee \cdots \vee t_n \neq \epsilon}{unionRec(\{A_1\ \cdots\ A_n,\ convSeq(x\ \alpha)\})\quad inapplicableScopeError}\ \textit{prefix-sequence-error}$$

Rewrite 5: Prefix Sequences Unsafe

4.3.2 Union Definition

To define the productions of $unionRec(O)$, we first initialize it with the empty production: $unionRec(O) \to \epsilon$. Then any production belonging to a symbol in O is copied and made right-recursive according to Rewrite Rules 6. As discussed before, there may be aliases, in which case the first production symbol is not a regular expression. Such aliases are followed until the symbol with actual productions is reached, as described in rule *define-alias-unionRec*(O).

$$\frac{A \in O \quad A \to x\ \alpha}{A \to x\ \alpha \quad unionRec(O) \to x\ \alpha\ (\langle\epsilon\rangle unionRec(O))}\ \textit{define-unionRec}(O)$$

$$\frac{A \in O \quad A \to A_1 \ \cdots\ A \to A_n \quad A_n \to x\ \alpha}{A \to A_1 \ \cdots\ A \to A_n \quad A \to x\ \alpha \quad unionRec(O) \to x\ \alpha\ (\langle\epsilon\rangle unionRec(O))}\ \textit{define-alias-unionRec}(O)$$

Rewrite 6: Union Definition Rules

After defining all productions of $unionRec(O)$ this way, we want to make sure it contains no two consecutive non-terminals. This is done according to the Rewrite Rules 7.

$$\frac{A \to \alpha\ (\langle t\rangle A)\ (\langle t\rangle B)\ \beta}{A \to \alpha\ (\langle t\rangle unionRec(\{B, C\}))\ \beta}\ \textit{merge-consecutive}$$

$$\frac{A \to \alpha\ (\langle\epsilon\rangle A)\ /x{+}\epsilon/\ (\langle\epsilon\rangle B)\ \beta}{A \to \alpha\ (\langle\epsilon\rangle unionRec(\{B, C, convSeq(x)\}))\ \beta}\ \textit{merge-regex-consecutive}$$

Rewrite 7: Consecutive Merging

These transformations use new $unionRec(O')$ symbol to simulate a sequence of non-terminals. This language will contain phrases from the original sequence, as well as phrases from many other – newly allowed – sequences. For the second transformation rule, we once again use TCNFAs to detect whether a regular expression accepts an empty word, and use our function D to obtain a regular expression matching only the non-empty words. When performing the pattern matches, the scopes are ignored. If the scopes happen to not match the pattern, an error is generated but the pattern is still applied, as illustrated in Rewrite Rules 8. This defined productions should now match the specification of a given $unionRec(O)$ symbol.

$$\frac{A \to \alpha\ (\langle t\rangle A)\ (\langle t'\rangle B)\ \beta \quad t \neq t'}{A \to \alpha\ (\langle\epsilon\rangle unionRec(\{B, C\}))\ \beta \quad inapplicableScopeError}\ \textit{merge-consecutive}$$

$$\frac{A \to \alpha\ (\langle t\rangle A)\ /x{+}\epsilon/\ (\langle t'\rangle B)\ \beta \quad t \neq \epsilon \vee t' \neq \epsilon}{A \to \alpha\ (\langle\epsilon\rangle unionRec(\{B, C, convSeq(x)\}))\ \beta \quad inapplicableScopeError}\ \textit{merge-regex-consecutive-error}$$

Rewrite 8: Consecutive Merging Unsafe

Finally the created productions are deduplicated, removing any productions that define an equivalent language as another production for this same symbol. This is done based on the same production equivalence check as used in grammar deduplication. In summary, $unionRec(O)$ defining consists of 3 steps:

1. Copy productions from the sources
2. Combine consecutive non-terminals
3. Deduplicate productions

When creating new $unionRec(O)$ symbols in the grammar, transformation rules are used to obtain a normal form. These transformations ensure that the specified language remains unchanged, while decreasing the number of symbols that have to be defined. Rewrite Rules 9 contains these simplification rules.

$$\frac{unionRec(\{unionRec(O), o_1, \ldots, o_n\})}{unionRec(O \cup \{o_1, \ldots, o_n\})}\ flatten$$

$$\frac{unionRec(\{A, o_1, \ldots, o_n\}) \quad A \to B}{unionRec(\{B, o_1, \ldots, o_n\}) \quad A \to B}\ resolve\text{-}alias$$

Rewrite 9: Simplify Unions

4.3.3 Closing Definition

Defining of $closed(s, c)$ symbol productions is done similar to defining $unionRec(O)$, but requires additional transformations to reach the desired form. Again, the symbol is initialized to have an empty production: $closed(s, c) \to \epsilon$. Then for any production of s and any production of c, their right-recursive sequence can be added as described in Rewrite Rule 10.

$$\frac{s \to x\ \alpha \quad c \to y\ \beta}{s \to x\ \alpha \quad c \to y\ \beta \quad closed(s,c) \to x\ \alpha\ y\ \beta\ (\langle\epsilon\rangle closed(s,c))}\ define{-}closed(s,c)$$

Rewrite 10: Closing Definition Rules

Again, if s or c is an alias, their alias sequence will be followed until a prefixed production is reached similar as for $unionRec(O)$ symbols. These initially defined productions are then deduplicated, the same way $unionRec(O)$ symbol productions were deduplicated in the end. We then combine consecutive non-terminal symbols, the same way as was done for $unionRec(O)$ symbols.

Next, direct left-recursion is removed according to Rewrite Rule 11.

$$\frac{A \to /x{+}\epsilon/\ (\langle\epsilon\rangle B)\ \cdots\ /z{+}\epsilon/\ (\langle\epsilon\rangle A)\ \alpha}{A \to x\ (\langle\epsilon\rangle B)\ \cdots\ /z{+}\epsilon/\ (\langle\epsilon\rangle A)\ \alpha \quad A \to \alpha}\ remove\text{-}direct\text{-}recursion$$

$$\vdots$$

$$A \to z\ (\langle\epsilon\rangle A)\ \alpha$$

Rewrite 11: Remove Direct Recursion

This rule is applied to any sequence of nullable production components being followed by the production symbol itself. Every non-terminal is assumed to be nullable, and whether a regular expression is nullable can be checked using its TCNFA. For every nullable regular expression, a production where this

regular expression is the first of the production and is not nullable is added. Finally a production is added of the remainder that can follow the self-reference. If the scopes do not match this pattern, we apply Rewrite Rule 12 instead, and generate an appropriate error indicating that not all tokenizations could be preserved.

$$\frac{A \to /x{+}\epsilon/\ (\langle t_1 \rangle B_1)\ \cdots\ /z{+}\epsilon/\ (\langle t_n \rangle A)\ \alpha \quad t_1 \neq \epsilon \vee \cdots \vee t_n \neq \epsilon}{\begin{array}{l} A \to x\ (\langle \epsilon \rangle B_1)\ \cdots\ /z{+}\epsilon/\ (\langle \epsilon \rangle A)\ \alpha \quad A \to \alpha \quad \mathit{inapplicableScopeError} \\ \vdots \\ A \to z\ (\langle \epsilon \rangle A)\ \alpha \end{array}}\ \textit{remove-direct-recursion-error}$$

Rewrite 12: Remove Direct Recursion Unsafe

We then partially remove redundant closing expressions as describe in Rewrite Rule 13. This deals with potentially unnecessary complexity that is added by how the $closed(s, c)$ productions are defined.

$$\frac{closed(s, convSeq(x)) \to \alpha\ x\ s\ x\ (\langle \epsilon \rangle closed(s, convSeq(x)))}{closed(s, convSeq(x)) \to \alpha\ x\ (\langle \epsilon \rangle closed(s, convSeq(x)))}\ \textit{partially-remove-redundancy}$$

Rewrite 13: Partially Remove Redundancy

We can safely remove s and x here, because we know that $closed(s, convSeq(x))$ can match any number of these. Therefore removing the requirement of matching it at least once, will only broaden the defined language. Moreover, all $closed(s, c)$ expressions we define will be such that $c = convSeq(x)$ for some regular expression x. To generalize this rule, we do not actually check whether the exact regular expression x occurs twice as specified in the rewrite rule, but instead check whether the two regular expressions define an equivalent language by making use of their TCNFAs. This transformation is used to simplify the productions, and prevent the algorithm from defining unnecessary extra symbols and productions later. Productions matching this shape are commonly created as a result of the overlap transformation rule, which is the next applied rule described by Rewrite Rule 14. This is an important step, combining overlapping regular expressions that would cause non-determinism.

$$\frac{A \to /x{+}z/\ \alpha\ w\ \gamma \quad A \to /y{+}z/\ \beta\ w\ \gamma}{A \to /x{+}y{+}z/\ unionRec(\{convSeq(\alpha\ /(>w)/), convSeq(\beta\ /(>w)/)\})\ w\ \gamma}\ \textit{combine-overlap}$$

Rewrite 14: Combine Overlapping Productions

In order to generalize this rule, we detect overlap by using TCNFAs of the regular expressions to calculate the product automaton and check if its language is non-empty. When talking about a product automaton of TCNFAs, we refer to an automaton that contains words included in both automata and merges the corresponding tags, similar to how tags are merged in a TCNFA concatenation automaton. If we are interested in an automaton consisting of words and tag assignments that are shared exactly between two automata, we call this a strict product automaton. Here we are, however, interested in detecting overlap modulo tags and thus care for the non-strict product automaton. Moreover, we do not merely check whether there are words that are contained in both regular expressions, but instead check whether both regular expressions could apply at once. This is not only the case when they match the same exact word,

but also when a word matched by one of the expressions forms a prefix of a word matched by another expression. Therefore to check if two regular expressions x and y overlap, we compute the product automata for $/x.*/$ and y as well as x and $/y.*/$ to perform non-empty language intersection checks on. Instead of applying this rewrite rule on two productions at a time, all overlapping productions are collected at once and reduced to a single production. When doing this, it is important to realize that this notion of overlap is not transitive. Consider regular expressions $x = /ab/$, $y = /a/$, and $z = /ac/$. We can see that x and y overlap, since they can both start a match when encountering ab, similarly y and z overlap since they can both start a match when encountering ac, but x and z do not overlap and can never both start a match at the same index for any given input. Therefore when collecting all overlapping expressions, overlap with any expressions in the overlap class has to be checked, rather than relying on a single expression to be representative of the class when checking for inclusion. Something else that is not captured in our schematic rewrite rule overview, is how $convSeq(\alpha)$ is defined in a way that α always start with a regular expression. In case $\alpha = B_1 \cdots B_n\ w'\ \eta$ for non-terminal symbols B_1 to B_n, we replace $convSeq(\alpha)$ by $unionRec(\{B_1, \ldots, B_n, convSeq(w'\ \eta))\}$. In this case we end up with nesting in our unions, such as $unionRec(\{unionRec(\{B_1, \ldots, B_n, convSeq(w'\ \eta)\}), convSeq(\beta)\})$. If any of B_1 to B_n contained non-empty scopes in the original component sequence, an error is generated since these scopes have to be dropped. It is important to note that the transformations applied so far ensure that for any two productions, a common $w\ \gamma$ suffix exist. For some productions this shared suffix may be longer than for others, but there is always a common shared suffix. This transformation rule always uses the longest shared suffix. Adding this w as a lookahead for the component sequences prevents consecutive non-terminal symbol merging when the union symbol they belong to is defined. This lookahead is also the reason the previous transformation rule is often applicable. Similarly, we can also keep the longest shared prefix when applying this rule, since overlapping symbols will be merged in following iterations anyhow. We can also decide that the suffix should not end on a regular expression but may end on any symbol. When this is done, we do not add any lookahead to the sequences in the union, and the shared non-terminal should be added to the union itself in order to not have two consecutive non-terminals. This makes the grammar less strict, which can cause more non-determinism. In our test-cases this did however not cause any additional non-determinism, but did greatly reduce generation times. This will be demonstrated in our results.

Next, the remaining redundant sequences are fully removed according to Rewrite Rule 15. This transformation is similar to the one before, but does not guarantee the resulting productions to have a common suffix starting with a regular expression. This was an important property before applying the overlap merging but is no longer relevant.

$$\frac{closed(s, convSeq(x)) \rightarrow \alpha\ s\ x\ (\langle\epsilon\rangle closed(s, convSeq(x)))}{closed(s, convSeq(x)) \rightarrow \alpha\ (\langle\epsilon\rangle closed(s, convSeq(y)))}\ \textit{remove-redundancy}$$

Rewrite 15: Remove Redundancy

The remaining productions may have an arbitrary length, so they have to be changed to be of a fixed length using Rewrite Rules 16. Every production that is not of the shape $A \rightarrow x\ (\langle\epsilon\rangle A)$ or $A \rightarrow x\ B\ y(\langle\epsilon\rangle A)$ will be converted to be of the shape $A \rightarrow x\ B\ y(\langle\epsilon\rangle A)$. We know this is the case because no consecutive non-terminals can appear in any production after the above rewrite steps have been performed, and all productions start with a regular expression. Therefore any remaining production are either of the desired shape, or match this pattern.

$$\frac{A \to x\ \alpha\ y\ A \quad \alpha \neq B}{A \to x\ convSeq(\alpha\ /(>y)/)\ y\ A}\ split\text{-}sequences$$

Rewrite 16: Split Sequences

Finally we add a closing expression to our hierarchical productions as described by Rewrite Rule 17. This ensures that any production reachable from a $closed(s, c)$ production also is a $closed(s', c')$ production and thus has the desired shape.

$$\frac{A \to x\ (\langle t \rangle B)\ y\ A}{A \to x\ (\langle t \rangle closed(B, convSeq(/(>y)/)))\ y\ A}\ carry\text{-}closing$$

Rewrite 17: Carry Closing

One last issue that possibly remains, is possible presence of left-recursive cycles. The first regular expression of production may be nullable, for instance in the form of a lookahead. Due to the direct recursion removal, it is not possible that there is any direct recursion, but there may still be indirect recursion. This is checked by following any paths that may not consume any characters, and if this at any point arrives back at $closed(s, c)$, a cycle exists. In this case, our approach offers no good way of dealing with this, and instead generates an error and removes the recursion by narrowing the language as described by Rewrite Rule 18.

$$\frac{A \to /x{+}\epsilon/\ B_1\alpha \quad B_1 \to /y_1{+}\epsilon/\ B_2\ \gamma_1 \quad \cdots \quad B_n \to /y_n{+}\epsilon/\ A\ \gamma_n}{A \to /x/\ B_1\ \alpha \quad B_1 \to /y_1{+}\epsilon/\ B_2\ \gamma_1 \quad \cdots \quad B_n \to /y_n{+}\epsilon/\ A\ \gamma_n \quad cycleError}\ remove{-}recursion$$

Rewrite 18: Remove Recursion

In summary, $closed(s, c)$ defining consists of 10 steps:

1. Merge productions from the sources
2. Deduplicate productions
3. Combine consecutive non-terminals
4. Remove direct self-recursion
5. Partially remove redundant closing sequences
6. Combine overlapping productions
7. Fully remove redundant closing sequences
8. Split sequences into sub-symbols
9. Carry closing expressions
10. Check indirect self-recursion

4.3.4 Termination

It is not obvious this algorithm will terminate. In fact, in its current state, it might not terminate at all. Both $unionRec(O)$ and $closed(s, c)$ symbol structures are defined in terms of other symbols, and can not but can not directly contain nested symbols of their own type. $closed(s, c)$ may contain a $unionRec(O)$ for its argument s or c, but no $unionRec(O)$ can contain another $unionRec(O') \in O$ or $closed(s, c) \in O$. As such, when only dealing with these custom symbols, only a finite number of possible combinations to construct such symbols exists, and hence termination would be guaranteed. This however does no longer hold with the introduction of $convSeq(\alpha)$ symbols. The length of every sequence α is bounded in terms of the maximum length of a production in the original grammar, but its nesting complexity is not bounded. Consider grammars with if-statements for example. After an if-statement, we can usually see an optional else-statement. The current conversion algorithm essentially creates a grammar that encodes how many if-statements have been seen in its production rules, such that at most an equivalent number of else-statements may be seen afterwards. This is encoded in sequences such as: $convSeq(/else/\ unionRec(\{Stmt, convSeq(/else/\ Stmt\ /(>end)/)\})\ /(>end)/)$. The algorithm will keep generating new symbols, each of which deepen the nesting more. This is a problem for which this thesis provides no proper fix, and is left for future research. A quick fix has been implemented however, where the algorithm checks for nested structures whenever any new $convSeq(\alpha)$ created, and generalizes sequence when nesting is found. Detecting nested patterns is done by considering the top-level sequence α, searching through any nested $convSeq(\beta)$ components within α, and checking whether their shape is equivalent when only considering the language of regular expressions of the sequences. If such a nesting is detected in a non-terminal B, the new $convSeq(\alpha)$ definition will be relaxed according to Rewrite Rule 19.

$$\frac{convSeq(\alpha\ B\ \beta)}{unionRec(\{convSeq(\alpha), B, convSeq(\beta)\})}\ \textit{relax-sequence}$$

Rewrite 19: Relax Sequence

By limiting what custom symbols can be generated, we cause only a finite number of custom symbols to be obtainable. Since the algorithm can only define a finite number of these, it must eventually terminate. This however only holds if all recursive structures are properly detected, and is even then more of a theoretical guarantee than a practical guarantee. The output grammar might require an exponential number of symbols in relation to the size of the input grammar. And as such, there is a lot of room for improvement, as we will see in Section 6.5.

After this main loop has stopped, a little bit of cleanup is performed. Many of the non-terminal symbols in the grammar are used within custom symbol definitions, but are not actually reachable within the grammar. These non-terminals and their productions can now safely be removed from the grammar. Additionally, the grammar may still have some indirections in the form of alias symbols, these can also be removed from the grammar. In order to remove these, every occurrence of an alias symbol should be replaced by the symbol that it is aliasing. After this cleanup, the grammar is a proper LCG.

4.4 Determinism Check

After performing all these conversion steps, a proper LCG is obtained. We assume no errors to be generated during this process, and if there are, the grammar should be modified to resolve these. Having an error free LCG does, however, not ensure that syntax highlighters using this grammar now highlight according to our specification grammar. Even though the LCG guarantees to include the original tokenization, more valid tokenization for phrases in the original language may have been added, hence the grammar might have become ambiguous. In this case, our LCG obtains multiple tokenizations for a given input, and it is no longer known which of these is the one intended. Detecting of ambiguity in CFGs in general is undecidable and is therefore something we can not do. Fortunately we are interested in a stronger property than the

grammar being unambiguous, we need it to be deterministic. Even if the grammar defines only a single tokenization per input, tokenizers will make deterministic choices that might not obtain this single valid tokenization. Therefore we are interested in checking whether tokenizers only have a single valid option at every critical area in the grammar. This would ensure that their greedy tokenization is indeed a correct tokenization. For this, 4 types of choices made by tokenizers have to be considered:

- Which of the active patterns should be applied?
- Should the currently active patterns become inactive, returning to the previous patterns?
- How many characters should be matched by the matching regular expression?
- Which tokenization of the matching regular expression should be used?

For each of these questions, tokenizers make a greedy choice that could lead to an incorrect tokenization. To choose a pattern to apply, the fist pattern whose regular expression matches a prefix of the remainder of the input is chosen. This requires the patterns to be sorted, in order to have a well-defined notion of "first". Moreover, most tokenizers do not check whether the regular expression match is empty or not. This allows us to use lookaheads that do not capture any of the input to start a new rule. It, however, also allows for loops that never consume any input and thus never terminate. This is already taken care of during the conversion steps, by explicitly checking for cycles and removing them. To decide whether to deactivate a set of patterns, different tokenizers provide different degrees of control. This choice is generally based on whether a regular expression matches a prefix of the remainder of the input. The various levels of control come down to choosing whether deactivation of patterns has priority, or some other pattern in the active patterns has priority. This is only a priority, and thus the choice is ultimately still greedy and deterministic. The two remaining choices are actually results of a single choice made by the tokenizers: what path in the regular expression's NFA should be taken. Regular expression engines generally take the first path through which an accepting state is reached. The number of characters consumed by this is therefore not something the tokenizer directly chooses, and is not necessarily the longest nor shortest possible match. Multiple paths consuming the same number of characters might exist with different tokenizations. Since the tokenizer just chooses an essentially arbitrary path amongst these, an arbitrary tokenization is also chosen. We call a LCG without any of these types of choices, a choice-free LCG.

We define the following lemma regarding these choices:

Lemma 4. *If a LCG is choice-free, it defines at most 1 tokenization per word.*

This can be argued by considering the fixed-point equations that define the language of a LCG. With some inductive reasoning, we can show that the language for any production component, component sequence, and non-terminal symbol only has a single tokenization per word. The language of a regular expression component specifies only a single tokenization per word if the regular expression is not tag-ambiguous by definition. Additionally, if there is no choice for number of characters to be matched by a regular expression per word, no match in the language forms the prefix of another match in the language. Besides containing only a single tokenization per word, we show that in the language of a component sequence if a match w is a prefix of another match y, the remainder of y after removing prefix x is also in the language. The language of a component sequence $\mathit{regex}(x)\ \mathit{ref}(A, \epsilon)$ satisfies these properties, under the assumption the language of A does. No matches in $\mathcal{L}(x)$ forming a prefix of another match in $\mathcal{L}(x)$ is essential for this. Similarly, the language of a sequence $\mathit{regex}(x)\ \mathit{ref}(B, t)\ \mathit{regex}(y)\ \mathit{ref}(A, \epsilon)$ satisfies these properties, under the assumption the language of B and A do. For this to hold for sub-sequence $\mathit{ref}(B, t)\ \mathit{regex}(y)$, it is essential that no match of any first regular expression of B forms a prefix of a match of y or vice versa, as is the case if the LCG is choice-free. Lastly, if all these properties are met for every possible component sequence, and all alternations start with regular expressions for which no match forms the prefix of another match, the properties also hold for the language of any non-terminal symbol. Therefore the properties ultimately hold for the language of the choice-free LCG itself.

For each of these possible choices, we will perform a check on our grammar to see whether only a single choice exists. The shape conversion step already ensures that only a single active pattern per non-terminal is applicable at a time. Therefore no check has to be performed to ensure only a single active pattern can be applied. We do however have to perform a check to ensure no active pattern applies while a corresponding

closing expression also applies. Hence for every production $A \to x\ B\ y\ A$ and production $B \to z\ \alpha$ we check for overlap between y and z. This overlap check is performed the same way overlap was detected within the shape conversion step. If overlap between such regular expressions y and z is detected, a corresponding error is generated since we can not guarantee a correct tokenization for all inputs.

The two mentioned regular expression choices are results of the same greedy choice made by a tokenizer, but should be considered independently nevertheless. One could check whether only exactly one path exists for the NFA of a regular expression for every input, but this is a stronger condition than what we are interested in. To check whether only a single valid answer exists for the number of characters to be matched by a regular expression x, we check whether any word exists in both x and $/x.^+/$. This is done by creating the TCNFAs for both expressions, and checking whether the product automaton is empty. If this product is not empty, it is possible to encounter an input for which two different length prefixes are both matched by the same regular expression. In this case the tokenizer has to make a greedy choice, which might not be the choice that leeds to a correct tokenization where no input has to be skipped. Hence if such a regular expression is detected, a corresponding error is generated.

Finally, a single regular expression might be ambiguous and contain multiple tokenizations for the same input. In this case the grammar is fundamentally tag-ambiguous, and as such it is not the deterministic nature of a tokenizer that forms a problem, but it is really the grammar itself. Note that these types of ambiguities may be created by our conversion process, even if the input grammar was not ambiguous. These ambiguity cases might also exist in the grammar in ways that are not contained to a single regular expression, but these cases are all covered by the stronger determinism constraints. Fortunately presence of tag-ambiguity within regular expressions is decidable.

To calculate whether a regular expression x is tag-ambiguous, we first obtain x's TCDFA d. Since d is deterministic, only a single path per input exists. This input is a combination of characters and the tags (tokens) belonging to each character. The tags information can then be removed from d resulting in a TCNFA n, simply by replacing every tag set in a given transition by an empty set. If x contains multiple tag assignments for the same input, n must contain multiple paths for the same input. Hence our problem is reduced to detecting whether multiple accepting paths exist in a NFA that obtain the same word. This is known a ambiguity detection in NFAs, for which any known algorithm can be used [17]. We have reasoned that if x is tag-ambiguous, n must be ambiguous. But for this problem to be fully reduced to ambiguity detection, we also have to reason that if x is not tag-ambiguous, n is also not ambiguous. Assume this is not the case; x is not tag-ambiguous yet n is ambiguous. Then two paths matching the same input sequence exist in n, but because x is not tag-ambiguous, these paths have the same tokenization in d. This means d must not have been deterministic, and hence we reached a contradiction. Therefore n must be unambiguous if x is not tag-ambiguous.

When we detect a regular expression to be tag-ambiguous, a corresponding error is generated. If no errors are generated in this process nor the CG conversion process, we know that tokenizers will always produce the correct tokenizations when tokenizing according to the given grammar.

4.5 Mapping Lexing Conversion Grammars to Industry Formats

We now a proper LCG, and know that it is deterministic. The final step that has to be taken is converting this grammar to the desired LG format. This is mostly straight forward, but does still require us to perform a couple of interesting steps. Additionally, we will convert the LCG to a couple of intermediate formats that encodes a LG architecture without using the exact expected syntax. This allows us to share some of this structure between multiple different LGs. We introduce two intermediate models: scope grammars, and the PDA grammars. We will provide an approach for mapping from LCGs to scope grammars, and an approach for mapping from scope grammars to PDA grammars. Scope grammars can then be mapped to TextMate grammars, while PDA grammars can be mapped straightforwardly to Monarch, Ace, and Pygments grammars.

4.5.1 Scope Grammars

Scope grammars consist of a start symbol and a list of patterns per grammar symbol. Patterns come in three forms: inclusion of other symbol patterns, simple patterns, and hierarchical patterns. A simple pat-

tern specifies a regular expression to be matched, and what categories to assign per capture group. A hierarchical pattern consists of an opening regular expressions with associated categories, a symbol with an optional category for this symbol, and a closing regular expression with associated categories. Here regular expressions are no longer tagged with categories directly, but instead with indices of capture groups. Every capture group index can then be used to obtain the corresponding category from the list of categories. More formally, a scope grammar is defined by a tuple (V, Σ, T, R, s), where:

- V is a finite set of non-terminal symbols
- Σ is a finite set of terminal symbols
- T is a finite set of tags
- $R : V \to \mathcal{P}_s(V, \Sigma, T)^*$ is a mapping from symbols to pattern lists
- $s \in V$ is the start symbol

We then define the pattern universe $\mathcal{P}_s(V, \Sigma, T)$ as the minimal set satisfying:

$$\begin{aligned}
\mathit{include}(v) &\in \mathcal{P}_s(V, \Sigma, T) && \text{for } v \in V \\
\mathit{simple}((r, t)) &\in \mathcal{P}_s(V, \Sigma, T) && \text{for } r \in \mathcal{R}(\Sigma, \mathbb{N}), t \in T^* \\
\mathit{hierarchical}((r_b, t_b), (v, s), (r_e, t_e)) &\in \mathcal{P}_s(V, \Sigma, T) && \text{for } r_b, r_e \in \mathcal{R}(\Sigma, \mathbb{N}), t_b, t_e \in T^*, v \in V, s \in T \cup \{\epsilon\}
\end{aligned}$$

To ensure the capture group index to category mapping is well-defined, we also have some additional requirements for every regular expressions: natural numbers in tags must be smaller or equal to the number of categories, no tags may be present within lookarounds, and tags must be assigned incrementally. This last constraint is intended to correspond to how regular expression engines number capture groups. This is done by an in order tree walk on the syntax of the regular expression, assigning each capture group an identifier starting at one incrementing by one for every next group. Finally we require that the $\mathit{include}(v)$ patterns do not reference each other recursively.

We will define the semantics of such a grammar directly in terms of the tokenization it provides for a given input. These semantics will match those of LGs, such that exactly a single tokenization is created for every possible input sequence. Our semantics map a given input to a tokenization, which specifies per character of the input what the corresponding scope is. To define this function $\mathcal{T}_c : \Sigma^* \to (T^*)^*$, we rely on several other recursive functions. We will first define a function A_s that can be used to apply regular expressions and assign their corresponding tokenization. This function will have several arguments: The tokenization state in $(\Sigma^* \times (T^*)^*) \times \Sigma^*$ consisting of the already tokenized input and the remainder of the input, the current scope in T^*, and the regular expression and categories to apply in $\mathcal{R}(\Sigma, \mathbb{N}) \times T^*$. The output will be a new tokenization state in $(\Sigma^* \times (T^*)^*) \times \Sigma^*$. Therefore we can specify the signature as follows: $A_s : ((\Sigma^* \times (T^*)^*) \times \Sigma^*) \times T^* \times (\mathcal{R}(\Sigma, \mathbb{N}) \times T^*) \to ((\Sigma^* \times (T^*)^*) \times \Sigma^*)$. This function will itself rely on two helper functions to map the natural numbers to the corresponding categories. These functions can be defined as follows:

$$\begin{aligned}
A_s &: ((\Sigma^* \times (T^*)^*) \times \Sigma^*) \times T^* \times (\mathcal{R}(\Sigma, \mathbb{N}) \times T^*) \to ((\Sigma^* \times (T^*)^*) \times \Sigma^*) \\
A_s(((p, t), i), s, (r, c)) &= ((pw, t\ M_s(s, t', c)), i') && \text{where } ((p, \emptyset), (w, t'), (i', \emptyset)) \in \mathcal{L}(r) \wedge wi' = i \\
&= ((p, t), i) && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
M_s &: T^* \times \mathcal{P}(\mathbb{N})^* \times T^* \to (T^*)^* \\
M_s(s, t\alpha, c) &= (s \cdot E_s(t, c, 0)) \vdash M_s(s, \alpha, c) \\
M_s(s, \epsilon, c) &= \epsilon
\end{aligned}$$

$$\begin{aligned}
E_s &: \mathcal{P}(\mathbb{N}) \times T^* \times \mathbb{N} \to T^* \\
E_s(T, \epsilon, i) &= \epsilon \\
E_s(T, c\alpha, i) &= c \cdot E_s(T, \alpha, i + 1) && \text{where } i \in T \\
&= E(T, \alpha, i + 1) && \text{otherwise}
\end{aligned}$$

In this notation, the $\cdot$ operator represents sequences concatenation, and $\vdash$ represents sequence prefixing adding a single element to the start of another sequence. This function then simply says that if some match for the regular expression exists that starts at the remainder of the input to be tokenized, such a match is chosen. The tags of this match are then translated to the corresponding categories and combined with the provided scope. If the regular expression does not match, the tokenization state remains unmodified.

The tokenization function $\mathcal{T}_c$ will be an application of a generalized function $\mathcal{T}'_c$. This generalized function has multiple arguments: a tokenization state in $(\Sigma^* \times (T^*)^*) \times \Sigma^*$, a active scope state in $V \times T^*$, the scope deactivation regular expression and categories in $\mathcal{R}(\Sigma, \mathbb{N}) \times T^*$, and the pattern queue in $\mathcal{P}_s(V, \Sigma, T)^*$. The function in turn outputs a new tokenization state $(\Sigma^* \times (T^*)^*) \times \Sigma^*$. The idea is that this helper function tokenizes input until either the end of the input is reached, or the end pattern is matched. $\mathcal{T}_c$ and $\mathcal{T}'_c$ are defined as follows, using the grammar's tokenization rules R and start symbol s:

$$\begin{aligned}
\mathcal{T}_c &: \Sigma^* \to (T^*)^* \\
\mathcal{T}_c(i) &= t \qquad \text{where } ((i,t),\epsilon) = \mathcal{T}'_c(((\epsilon,\epsilon),i),(s,\epsilon),(/\mathbf{0}/,\epsilon),R(s))
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}'_c &: ((\Sigma^* \times (T^*)^*) \times \Sigma^*) \times (V \times T^*) \\
&\qquad \times (\mathcal{R}(\Sigma,\mathbb{N}) \times T^*) \times \mathcal{P}_s(V,\Sigma,T)^* \to (\Sigma^* \times (T^*)^*) \times \Sigma^* \\
\mathcal{T}'_c((p,\epsilon),(v,s),c,q) &= (p,\epsilon) \\
\mathcal{T}'_c(i,(v,s),c,q) &= o && \text{where } o = A_s(i,s,c) \wedge o \neq i \\
\mathcal{T}'_c(((p,t),a\alpha),(v,s),c,\epsilon) &= \mathcal{T}'_c(((pa,t \dashv s),\alpha),(v,s),c,R(s)) \\
\mathcal{T}'_c(i,(v,s),c,\mathit{include}(v')\beta) &= \mathcal{T}'_c(i,(v,s),c,R(v')\beta) \\
\mathcal{T}'_c(i,(v,s),c,\mathit{simple}(p)\beta) &= \mathcal{T}'_c(o,(v,s),c,R(v)) && \text{where } o = A_s(i,s,p) \wedge o \neq i \\
&= \mathcal{T}'_c(i,(v,s),c,\beta) && \text{otherwise} \\
\mathcal{T}'_c(i,(v,s),c,\mathit{hierarchical}(b,(v',s'),c')\beta) &= \mathcal{T}'_c(\mathcal{T}'_c(o,(v',ss'),c',R(v')),(v,s),c,R(v)) \\
& && \text{where } o = A_s(i,s,b) \wedge o \neq i \\
&= \mathcal{T}'_c(i,(v,s),c,\beta) && \text{otherwise}
\end{aligned}$$

In this notation, $\dashv$ is used to specify that one element is added to the end of a sequence. This function recurses until all characters in the tokenization state have been tokenized. For each symbol, the list of applicable patterns is loaded and tested one at a time. The inclusion pattern modifies the list of remaining patterns to be tested against. If such an inclusion pattern has a circular reference, this would recurse indefinitely, but this has been prohibited by the constraints for our grammar. When a simple pattern matches, it is used to update the tokenization state, and all patterns will be checked again on the next character to be processed. Finally, if a hierarchical pattern matches, the tokenization state is updated just like with the simple pattern, but we then start tokenizing with a new closing pattern and active scope state. This tokenization will resolve either when all characters are consumed, or when the closing pattern has been matched. The resulting tokenization state is processed with the original closing pattern and active scope state.

Mapping a LCG to a scope grammar is mostly straight forward, as was described in Section 2.5. A LCG only has three production forms, each of which can be mapped to a pattern that retains the corresponding semantics:

- $A \to \epsilon$ is implicitly present in the semantics of scope grammars, and can be skipped
- $A \to x\ (\langle\epsilon\rangle A)$ can be mapped to some pattern $(A, \mathit{simple}((y,t)))$
- $A \to x\ (\langle t\rangle B)\ y\ (\langle\epsilon\rangle A)$ can be mapped to some pattern $(A, \mathit{hierarchical}((x',t_x),(B,t'),(y',t_y))) \in R$

The patterns corresponding to a given production end up not containing the right-recursive self-reference to the production symbol. Since our grammars often include the same patterns for multiple symbols, we try to reuse these patterns to make the grammar shorter. For any component sequence α a dedicated symbol S_α

is defined, only specifying the pattern corresponding to α. Then for any production $A \to \alpha$ in the grammar, we add a corresponding pattern $(A, include(S_\alpha)) \in R$.

Despite the mapping being mostly straightforward, several minor discrepancies are left to be solved. One minor issue is that a LCGs may define a scope t of arbitrary length for a non-terminal B, while scope grammars can only apply a single scope per hierarchical pattern. This can be dealt with by performing a transformation rule on the LCG exhaustively, in order to remove any productions in which such a scope is present:

$$\frac{A \to x\ (\langle t_1 t_2 \alpha \rangle B)\ y\ (\langle \epsilon \rangle A) \quad A' \not\to}{A \to /(>x)/\ (\langle t_1 \rangle A')\ y\ (\langle \epsilon \rangle A) \quad A' \to x\ (\langle t_2 \alpha \rangle B)\ /(>y)/\ (\langle \epsilon \rangle A')}\ \textit{split-scopes}$$

This transformation maintains an equivalent language, and simply splits the scopes into single category scopes. The remaining discrepancies relate to regular expressions. The regular expression model we used is different from regular expressions used by tokenizers. Therefore we want to convert regular expressions in our format to equivalent expressions that translate straightforwardly to common regular expression syntaxes. There are three issues that have to be dealt with:

- A regular expression may make use of our binary lookaround operators, while mosts regex engines make use of unary operators. This can be dealt with using rewrites based on the following equivalences:
  - $\mathcal{L}(/x{>}y/) = \mathcal{L}(/x(>)y/)$
  - $\mathcal{L}(/x{\not>}y/) = \mathcal{L}(/x(\not>)y/)$
  - $\mathcal{L}(/y{<}x/) = \mathcal{L}(/(y<)x/)$
  - $\mathcal{L}(/y{\not<}x/) = \mathcal{L}(/(y\not<)x/)$
- A regular expression encodes scope tags directly in its syntax, while tokenizers rely on capture groups
- A regular expression may make use of the subtraction operator, which is not commonly supported by regex engines

Extracting scopes from a regular expression is quite straight forward, as was hinted at earlier. Here we formalize this by providing a recursive function E that operates on a regular expression and a category list and outputs a new regular expression only containing numbered capture groups together with a list of categories corresponding to every capture group. The fully definition of this function can be found in Appendix D. Since this function is rather trivial, we only provide a condensed overview below:

$$\begin{aligned}
E(\mathbf{0},\ s) &= (\mathbf{0},\ s) \\
&\vdots \\
E(r_1{+}r_2,\ s) &= (r_1'{+}r_2',\ s'') && \text{where } (r_1', s') := E(r_1, s) \wedge (r_2', s'') = E(r_2, s') \\
&\vdots \\
E((\langle t \rangle r_1), s) &= ((\langle |st| \rangle r_1'), s') && \text{where } (r_1', s') := E(r_1, st)
\end{aligned}$$

Then for a regular expression x, we can simply obtain a regular expression and category list pair by applying $E(x, \epsilon)$. Before doing this, all subtraction operators should be removed however.

This process is not trivial. We attempt to replace subtraction operators by negative lookaheads or negative lookbehinds. This approach is not guaranteed to be successful however, in which case this mapping process may yield a new error. Consider the regular expression $x = /[a{-}z]^*{-}word/$ specifying all alphabetic words, except *word*. Here the shorthand $[a{-}z]$ is used to describe the alternation of all alphabetic characters. A first attempt to get rid of the subtraction operator might be to use a negative lookbehind as follows: $x' = /[a{-}z]^*(word{\not<})/$. This expression specifies that it may see any word, as long as it does not end in *word*. This is not exactly the same however, since the check for *word* can start anywhere. It can start before the first character matched by $/[a{-}z]^*/$, on the first character as we intend, or after it. Consider

the input *word*, where the first character w has already been consumed. Then the remainder *ord* would be matched by x, but x' would not capture it, due to w being part of the prefix. If our original regular expression provides more context, it might be possible to detect that such cases can not happen however. Consider for instance $y = /A([a{-}z]^*{-}word)/$ where A is simply a literal uppercase character. Then we could define another regular expression $y' = /A([a{-}z]^*(word\not<))/$ that replaces the subtraction with a negative lookbehind. Now we see that the disallowed *word* capture could never start before the first match of $/[a{-}z]^*/$, since none of the characters in *word* are equal to A, thus they can not overlap this character. The only importance of this A prefix is that it provides information about the surrounding context that may occur, it is not important that this is part of the actual match itself. As such, the negative lookbehind that is commonly specified for identifier patterns has the same effect: $/[a{-}z]\not<([a{-}z]^*{-}word)/$ can be transformed to $/[a{-}z]\not<([a{-}z]^*(word\not<))/$ which does not disallow any additional words based on the context of the words. Another issue does remain however, where *aword* would be matched by y but not by y'. Since y' still only checks for whether the word does not end in *word*, this check can start after the first character of $/[a{-}z]^*/$ is matched. To prevent this, more context can be provided to the subtracted term. The contextual information present in concatenation distributes over subtraction, hence y can be transformed to $/(A[a{-}z]^*){-}Aword/$. The subtraction can be then be replaced by a negative lookbehind to obtain $y'' = /(A[a{-}z]^*)(Aword\not<)/$. In this expression the *Aword* match could never start after having performed the first match of $/[a{-}z]^*/$, because A has no overlap with this expression. Therefore these languages are in fact equivalent. Once again, the same would apply for a version where lookarounds are used: $\mathcal{L}(/[a{-}z]\not<([a{-}z]^*{-}word)/) = \mathcal{L}(/([a{-}z]\not<[a{-}z]^*)(([a{-}z]\not<word)\not<)/)$. This means that the common expression used to specify identifiers in Rascal grammars can perfectly be translated to a regular expression that does not make use of subtraction operators.

This idea is generalized to define an algorithm for automatically removing subtraction operators from regular expressions. This algorithm does not guarantee that the resulting regular expression has an equivalent language, but does generate errors if and only if the resulting regular expression does not define an equivalent language. This is done by making use of the expressions' TCNFAs to determine whether the language was maintained. The algorithm attempts to move as much of the contextual information surrounding a subtraction into the subtracted expression. This relies on several regular expression language equivalences. Appendix G provides an extensive set of axioms for these equivalences, but not all of these are important for this transformation. This transformation relies specifically on the following equivalences, where x, y, and z are arbitrary regular expressions:

- $\mathcal{L}(/(x{>}y){-}z/) = \mathcal{L}(/(x{>}y){-}(z{>}y)/)$, with symmetric versions for all other lookaround types
- $\mathcal{L}(/(x{-}z){>}y/) = \mathcal{L}(/(x{>}y){-}(z{>}y)/)$, with symmetric versions for all other lookaround types
- $\mathcal{L}(/(x{-}z)y/) = \mathcal{L}(/(xy){-}(zy)/)$, with a symmetric version for prefix concatenation
- $\mathcal{L}(/(x{-}y){-}z/) = \mathcal{L}(/x{-}(y{+}z)/)$

The algorithm implicitly uses these equivalences to extract information while ensuring language equivalence, and uses the subtraction to negative lookaround transformation whenever necessary to deal with sub-expressions. We define an inductive function $S : \mathcal{R}(\Sigma, T) \to \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathbb{B}$ that splits a regular expression in 4 parts, and a boolean specifying whether equivalence is maintained. More specifically, given a regular expression x, it outputs a lookbehind regular expression b, a match expression m, a lookahead expression a, and a subtraction expression s, together with a boolean *equal*. It ensures that the following properties hold on this output:

- $equal \Rightarrow \mathcal{L}(/m{-}s/) = \mathcal{L}(x)$
- $\mathcal{L}(m) = \mathcal{L}(/bma/)$
- If x does not contain any regular expression, *equal* is *true*
- None of the output expressions contain subtractions

The definition of this function will rely on a helper function $S_l : \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \to \mathcal{R}(\Sigma, T) \times \mathbb{B}$ which attempts to combine the four expressions b, m, a and s into an expression y either as

$y = /(\not>bsa)m/$ or $y = /m(bsa\not<)/$. This function attempts to ensure that $\mathcal{L}(y) = \mathcal{L}(m-s)$, and returns a boolean indicating whether this was achieved. Using this function, and the results $(b_1, m_1, a_1, s_1, e_1) = S(r_1)$ and $(b_2, m_2, a_2, s_2, e_2) = S(r_2)$ obtained for sub-expressions, we can inductively define the function S. See Appendix C for the full definition, only some representative cases are provided here:

$$
\begin{aligned}
S(a) &= (\epsilon, a, \epsilon, \mathbf{0}, true) \\
S(r_1 > r_2) &= (b_1, m_1 > a', a_1 > a', s_1, e_1 \wedge e_2 \wedge e) && \text{where } (a', e) = S_l(b_2, m_2, a_2, s_2) \\
S(r_2 \not< r_1) &= (b' \not< b_1, b' \not< m_1, a_1, s_1, e_1 \wedge e_2 \wedge e) && \text{where } (b', e) = S_l(b_2, m_2, a_2, s_2) \\
S(r_1 - r_2) &= (b_1, m_1, a_1, s_1 + s', e_1 \wedge e_2 \wedge e) && \text{where } (s', e) = S_l(b_2, m_2, a_2, s_2) \\
S(r_1 + r_2) &= (\epsilon, m_1' + m_2', \epsilon, \mathbf{0}, e_1 \wedge e_2 \wedge e_1' \wedge e_2') && \text{where } (m_1', e_1') = S_l(b_1, m_1, a_1, s_1), (m_2', e_2') = S_l(b_2, m_2, a_2, s_2)
\end{aligned}
$$

The helper function S_l can then be applied on the result of this inductive function to obtain the final expression, free of subtraction operators.

Only this subtraction removal step may cause problems regarding tokenization preservation. All other transformations ensure equivalence tokenizations, which could be shown straightforwardly using the defined semantics. As such, we specify the following lemma:

Lemma 5. *If all regular expressions in the mapping of a choice-free LCG to a scope grammar have their subtractions removed safely, all tokenizations are preserved*

4.5.1.1 Textmate grammars From such a scoped grammar, it is now quite trivial to map their constructs to corresponding TextMate constructs. When pretty printing a regular expression abstract syntax tree, some care has be taken regarding the precedences of operators however. One could simply surround every sub-expression with a non-capturing group, but this results in quite ugly expressions. For instance the regular expression $/if+else+for/$ would be stringified as *(?:(?:(?:if)+(?:(?:(?:el)s)e))+(?:(?:fo)r))*. Alternatively one could decide whether to put sub-expressions in non-capture groups, based on the precedence of their operator and the precedence of the parent operator, which results in cleaner expressions. This is a common problem when dealing with unparsing, and is therefore not unique to our research [18]. For our pretty printing, Oniguruma regular expression syntax is used [4]. The resulting grammar will perform syntax highlighting in a editor that supports TextMate grammars in accordance to the semantics defined for our scope grammars.

4.5.2 PDA Grammars

PDA grammars consists of a start symbol, and a list of patterns per grammar symbol. Patterns come in four forms: inclusion of other symbol patterns, simple patterns, push patterns, and pop patterns. The inclusion and simple patterns are the same as for scope grammars. The hierarchical pattern of scope grammars has been split into two rules, both of which extend the simple pattern behavior: a push pattern that adds a new symbol to the stack when its pattern matches, and a pop pattern that pops the current symbol from the stack when its pattern matches. The top symbol on the stack defines the patterns that are currently active. More formally, a PDA grammar is defined by a tuple (V, Σ, T, R, s), where:

- V is a finite set of non-terminal symbols
- Σ is a finite set of terminal symbols
- T is a finite set of tags
- $R : V \to \mathcal{P}_p(V, \Sigma, T)^*$ is a mapping from symbols to pattern lists
- $s \in V$ is the start symbol

We then define the pattern universe $\mathcal{P}_p(V, \Sigma, T)$ as the minimal set satisfying:

$$\begin{aligned}
\mathit{include}(v) &\in \mathcal{P}_p(V, \Sigma, T) && \text{for } v \in V \\
\mathit{simple}((r, t)) &\in \mathcal{P}_p(V, \Sigma, T) && \text{for } r \in \mathcal{R}(\Sigma, \mathbb{N}), t \in T^* \\
\mathit{pop}((r, t)) &\in \mathcal{P}_p(V, \Sigma, T) && \text{for } r \in \mathcal{R}(\Sigma, \mathbb{N}), t \in T^* \\
\mathit{push}((r, t), v) &\in \mathcal{P}_p(V, \Sigma, T) && \text{for } r \in \mathcal{R}(\Sigma, \mathbb{N}), t \in T^*, \in V
\end{aligned}$$

The requirements on regular expressions and *include* usage apply here as specified for scope grammars. Additionally, we require every regular expression to define a language where each character has exactly one associated tag. This can be achieved by making sure each character construction has exactly one tag construction as an ancestor, except for lookarounds. Finally, no pop patterns may be present in the rules of the start symbol.

The semantics of this format is provided in a form very similar to the semantics of a scope grammar in order to highlight the similarities and differences. The most notable difference is that PDA grammars always provide exactly one category per input character, rather than an arbitrary length scope. This is reflected in the regex application function A_p. The extra requirement we added on regular expressions is crucial for this. Helper function E_p has become a partial function now, but always provides an output when being called through A_p with a regular expression that satisfies the constraints. Below are the exact definitions:

$$\begin{aligned}
A_p &: ((\Sigma^* \times T^*) \times \Sigma^*) \times (\mathcal{R}(\Sigma, \mathbb{N}) \times T^*) \to ((\Sigma^* \times T^*) \times \Sigma^*) \\
A_p(((p, t), i), (r, c)) &= ((pw, t\ M_p(t', c)), i') && \text{where } ((p, \emptyset), (w, t'), (i', \emptyset)) \in \mathcal{L}(r) \wedge wi' = i \\
&= ((p, t), i) && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
M_p &: \mathcal{P}(\mathbb{N})^* \times T^* \to T^* \\
M_p(t\alpha, c) &= E_p(t, c, 0) \cdot M(\alpha, c) \\
M_p(\epsilon, c) &= \epsilon
\end{aligned}$$

$$\begin{aligned}
E_p &: \mathcal{P}(\mathbb{N}) \times T^* \times \mathbb{N} \to T \\
E_p(T, c\alpha, i) &= c && \text{where } i \in T \\
&= E_p(T, \alpha, i + 1) && \text{otherwise}
\end{aligned}$$

The tokenization function $\mathcal{T}_p$ now requires an extra argument, namely a token to be usd for skipped inputs. Once again, this function is defined in terms of a generalized function $\mathcal{T}'_p$. This generalized function operates on the tokenization state, requires the default token, a symbol to tokenize for, and a queue of patterns to try to apply. Tokenization finishes either when the end of the input stream is reached, or a pop pattern has been executed. Because the start symbol is not allowed to have any pop patterns, the input is guaranteed to always fully tokenize. $\mathcal{T}_p$ and $\mathcal{T}'_p$ are defined as follows, using the grammar's tokenization

rules R and start symbol s:

$$\begin{aligned}
\mathcal{T}_p &: \Sigma^* \times T \to T^* \\
\mathcal{T}_p(i,d) &= t && \text{where } ((i,t),\epsilon) = \mathcal{T}'_p(((\epsilon,\epsilon),i),d,s,R(s))
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}'_p &: ((\Sigma^* \times T^*) \times \Sigma^*) \times T \times V \times \mathcal{P}_p(V,\Sigma,T)^* \to (\Sigma^* \times T^*) \times \Sigma^* \\
\mathcal{T}'_p((p,\epsilon),d,v,q) &= (p,\epsilon) \\
\mathcal{T}'_p(((p,t),a\alpha),d,v,\epsilon) &= \mathcal{T}'_p((pa,td),d,v,R(v)) \\
\mathcal{T}'_p(i,d,v,pop(p)\beta) &= o && \text{where } o = A_p(i,p) \wedge o \neq i \\
&= \mathcal{T}'_p(o,d,v,\beta) && \text{otherwise} \\
\mathcal{T}'_p(i,d,v,include(v')\beta) &= \mathcal{T}'_p(i,d,v,R(v')\beta) \\
\mathcal{T}'_p(i,d,v,simple(p)\beta) &= \mathcal{T}'_p(o,d,v,R(v)) && \text{where } o = A_p(i,p) \wedge o \neq i \\
&= \mathcal{T}'_p(o,d,v,\beta) && \text{otherwise} \\
\mathcal{T}'_p(i,d,v,push(p,v')\beta) &= \mathcal{T}'_p(\mathcal{T}'_p(o,d,v',R(v')),d,v,R(v)) && \text{where } o = A_p(i,p) \wedge o \neq i \\
&= \mathcal{T}'_p(o,d,v,\beta) && \text{otherwise}
\end{aligned}$$

This definition is very similar to that of scope grammars. The main difference is that no end pattern is provided as a dedicated argument, but is instead present in the patterns queue in the form of a pop pattern. This makes the semantics simpler, but makes the grammar less declarative.

Converting a scope grammar to PDA grammar is mostly straight forward with the exception of dealing with scopes, as was mentioned in Section 2.5. Our mapping should operate on a scope grammar where every hierarchical pattern does not provide a scope for the internal symbol, and every regular expression has exactly one category applicable to every character match. Such a scope grammar is called a single category scope grammar. A single category scope grammar can be obtained by performing a transformation on the choice-free LCG that the scope grammar is derived from, prior to obtaining the scope grammar. Similarly, choice-free LCG resulting from these transformations is called a single category LCG. This transformation does not guarantee that the language is maintained, but does provide an error if any scope data had to be dropped. Assuming we have a scope grammar adhering to these constraints, the inclusion and simple patterns can remain the same for the corresponding PDA grammar. Every hierarchical pattern $hierarchical(b,(B,\epsilon),e)$ can be mapped to a push pattern $push(b,B_e)$. Here B_e represents a new symbol in the grammar. This symbol is then defined to have exactly two patterns in R: $R(B_e) = pop(e)\ include(B)$. This mapping will ensure that the resulting PDA grammar provides the same tokenization for all inputs as the simplified scope grammar does.

Obtaining a single category scope grammar is harder than this mapping process. First off, there are multiple choices for converting a category sequence α to a single category. The most obvious choice would be to use the last category in the sequence, and drop the first categories. We could however also merge all categories into a single category, such that less data is lost. Within our formal model no assumptions are made on the structure of categories, and thus we can not manipulate them. But we do know that in practice, they are dot separated sequences of category components. This means categories as essentially sequences of category components, and scope are sequences of these sequences. Hence a scope sequence of sequences, could be flattened to a single sequence of category components, which in term forms a single category. One problem with this merging technique is that the semantics of a conversion grammar are such that an infinite number of scopes can appear at runtime. When trying to convert such a grammar to one using only single category scopes, the grammar would have to encode an infinite number of possible scopes all as dedicated categories, leading to an infinite grammar. therefore, rather than treating a category as a sequence of components, it can be considered to be a set of components. In this case merging could consist of taking the union of all categories. Only a finite number of such sets exists when expressed in terms of category components in the original grammar, and thus the resulting grammar will also be finite. To make sure the category components are properly treated as sets when performing equivalence checks, the

components should be sorted to be in some consistent order in the sequence. We assume scopes to always be non-empty, which can be achieved by prefixing a default category to the scope, provided as an argument to the transformation. Regardless of the category merging process chosen, the transformation process of the grammar remains the same. We assume some category merging function *merge* to exist, that operates on two categories at a time and obtains a single category. We will first consider how a regular expression makes sure every character receives a single category. This can be done with a recursive function M, that operates on a regular expression and a category. Appendix E contains a full definition of the function. Below is a condensed overview of the function for an input expression x built up from at most the sub-expressions r_1 and r_2:

$$
\begin{aligned}
M(\mathbf{0},\ c) &= \mathbf{0} \\
&\vdots \\
M(a,\ c) &= (\langle c \rangle a) \\
&\vdots \\
M(r_1{+}r_2,\ c) &= r_1'{+}r_2' && \text{where } r_1' = M(r_1, c) \wedge r_2' = M(r_2, c) \\
&\vdots \\
M(r_1{>}r_2,\ c) &= r_1'{>}r_2 && \text{where } r_1' = M(r_1, c) \\
&\vdots \\
M((\langle t \rangle r_1), c) &= M(r_1, c') && \text{where } c' = \mathit{merge}(t, c)
\end{aligned}
$$

This function essentially carries the category to the character leaves of the regular expression, applying the merge function on any intermediately encountered categories. There is some slight nuance with the lookaround expressions, where the category should only be carried towards the actual match and not the lookaround itself. This recursive function could be optimized to obtain simpler expressions with an equivalent language, by always first determining whether the regular expression x that it operates on contains any tag constructs. If x does not contain any tags, one can simply return $/(\langle c \rangle x)/$ regardless of the structure of x. Using this function, an input LCG can be converted to our desired grammar using a search function. We initialize a queue with custom symbols of the form A_c where A is a symbol in the original grammar, and c is a category. We initialize this queue to contain one symbol S_d where S is the start symbol of the input grammar, and d is a default category provided to the transformation. We then iterate over the queue, until it becomes empty. For every A_c in the queue, we copy every $A \to \epsilon$ production, adding $A_c \to \epsilon$ to our grammar. Every $A \to x\ A$ is copied as $A_c \to y\ A_c$ with $y = M(x, c)$. Finally $A \to x\ (\langle t \rangle B)\ y\ A$ is copied as $A_c \to z\ (\langle \epsilon \rangle B_{c'})\ w\ A$ where $c' = \mathit{merge}(c, t)$, $z = M(x, c)$, and $w = M(y, c)$. If $B_{c'}$ is not part of the generated grammar yet, $B_{c'}$ is added to the queue. After this process terminates, every regular expression in the resulting grammar will have a single category per character. When performing these merges, we can track whether any two categories neither of which are the default-category. If this is the case, it means that some tokenization data was lost. Otherwise the grammar preserves all tokenizations, when using the interpretation that the default category is equivalent to an empty scope.

Finally, if no errors are generated, the PDA preserves all tokenizations as more constructively expressed by the following lemma:

Lemma 6. *If LCG G_s can be converted to a single category LCG G_s' without merging two non-default categories, the derived PDA grammar preserves all tokenizations.*

4.5.2.1 Pygments Creating a Pygments grammar from a given PDA grammar is a trivial process. Each of the patterns in a PDA grammar has a straight forward counter part in a Pygments lexing grammar. Pygments grammars are however proper Python classes, and as such Python code would have to be generated. Our implementation instead generates a JSON grammar, and a helper Python class is provided to read these grammars and generate the Python lexing constructs at runtime. There is only one small problem we have to take care of in our initial mapping process: Pygments relies on an implicitly defined start symbol rather

than a explicitly defined start symbol. Instead of stating what the start symbol of a Pygments grammar is, the grammar is assumed to have a symbol called *root* representing the start symbol. Therefore we must rename what ever was specified to be the start symbol to *root*. Note that Pygments does not actually operate line by line, but this does not pose any problems. Line by line operation was never used as an assumption by our pipeline, but only used as a restriction, and thus these grammars operate as expected.

4.5.2.2 Ace Creating an Ace grammar from a given PDA grammar is almost identical to creating a Pygments grammar. Similar to Pygments grammars, Ace grammars are defined using a proper programming language, in this case JavaScript. The actual language specific concerns are all encoded using pure JSON. Therefore we can provide the actual grammar in JSON, and provide a template that merely loads this JSON definition and calls `this.normalizeRules();` to resolve any *include*(A) rules. Similar to Pygments grammars, the start symbol is defined implicitly. In this case the name of the start symbol should be *start*. Finally, Ace brings up one additional minor issue. The lines the tokenizer operates on omit the linefeed character. To correct for this, every regular expression has to be modified such that any character class x that includes the linebreak character become $/x+\$/$ where \$ is a special end-of-line matching construct that regex engines support.

4.5.2.3 Monarch Creating a Monarch grammar from a given PDA grammar is least trivial of all. Monarch grammars are encoded in JavaScript, just like Ace grammars. The actual language concerns are not encoded in pure JSON however, since javascript regex syntax is used for regular expressions. Therefore a small transformation function will have to be provided in javascript, that constructs regex instances from strings encoding these regular expressions. Additionally the flag *includeLF* should be provided to make sure line endings are included. The actual difficult part however is that regular expressions in Monarch are not allowed to have capture groups not match for some word. For instance the regular expression $/(\langle t_1\rangle a)+(\langle t_2\rangle b)/$ is not allowed, since either the capture group belonging to t_1 or the capture group belonging to t_2 will not be matched. Note that this is different from a capture group matching an empty string, therefore $/(\langle t_1\rangle a)(\langle t_2\rangle \epsilon)/$ is accepted by Monarch. Alternations are not fundamentally problematic though, as long as they occur within capture groups, such as: $/(\langle t\rangle a+b)/$. In order to deal with this, every regular expression is split up in a set of expressions that only contain alternations within tag statements, such that their union represents the original language. More formally, we define an inductive function $S : \mathcal{R}(\Sigma, T) \to \mathcal{P}(\mathcal{R}(\Sigma, T))$, such that for any regular expression x, we have $\mathcal{L}(x) = \mathcal{L}(/+_{y \in S(x)}\ y/)$. Appendix F contains a full definition of this function. Below is a condensed overview of the function for an input expression x built up from the sub-expressions r_1 and r_2, where $R_1 = S(r_1)$ and $R_2 = S(r_2)$:

$$
\begin{aligned}
S(\mathbf{0}) &= \{\mathbf{0}\} \\
&\vdots \\
S(r_1{+}r_2) &= R_1 \cup R_2 \\
S(r_1 r_2) &= \{r_1' r_2' \mid r_1' \in R_1, r_2' \in R_2\} \\
S(r_1^+) &= \{r_1\} \\
S(r_1{>}r_2) &= \{r_1'{>}r_2 \mid r_1' \in R_1\} \\
&\vdots \\
S((\langle t\rangle r_1)) &= \{(\langle t\rangle r_1)\}
\end{aligned}
$$

In this function, we make use of some of the constraints that hold for our regular expressions. We know that no tag constructs can appear within iteration constructs obtained by the regular expression conversion, and as such our definition for $S(r_1^+)$ is irrelevant. Similarly recursion stops at $S((\langle t\rangle r_1))$ since we know tag statements are not being nested for PDA grammars. Now using this function, every pattern can be mapped to a collection of patterns, which together define the same semantics as the original pattern.

4.6 Lookahead Improving

A full grammar conversion process has now been described. This process still contains several issues that will be discussed later. Most of these are left as future research, but a solution for one of these problems has already been explored. The problem in question, relates crucial follow data being lost by the conversion process. Consider a Rascal definition for identifiers, being translated to the regular expression $/([a-z]\not<[a-z]^+\not>[a-z])-y/$ where y is some alternation regular expression containing words such as *for* or if. As previously discussed, subtraction will have been replaced by negative lookbehinds, but this is harder to read and is not relevant for this problem given that their languages are equivalent. A pattern for this regular expression may be active together with the pattern $/for/$. Now when we encounter the text *for(*, the identifier expression will not match any prefix of this text, while $/for/$ does. However when encountering the text *fora*, both expressions match a prefix of this text. The identifier expression can match *fora* as a whole, while the $/for/$ expression simply matches *for*. Such non-determinism would frequently show up in our language, even though the original grammar likely won't have any ambiguity issues in these situations. A CFG in which $/for/$ occurs is likely to have a loop production that specifies *for* is always followed by an opening bracket. If the regular expression would have included this bracket, there is no overlap between this expression and the identifier expression. This expression would however not accurately encode the original language in most cases, because optional whitespace may be included between *for* and the bracket. If this is not allowed, our regular expression conversion step would indeed add the bracket in the same expression, and this problem does not show up. Unfortunately the same is not done when optional whitespace is allowed, since this optional whitespace can usually include linebreak characters, and tokenizers can not apply a single regular expression over multiple lines. However, instead of trying to make the regular expression match the entire entire construct including the bracket, we could hint at the characters that are allowed to follow the *for* keyword, using a lookahead. Let $[(]$ represent a regular expression matching the opening bracket. Then our $/for/$ regular expression could be changed to $/for>(\backslash\mathbf{n}+[(])/$ without affecting the language described by the grammar. This transformation prevents the described non-determinism. This transformation can be applied automatically, after the regular expression conversion step. For every regular expression, the regular expressions that may follow it can be calculated, and an alternation of these expression can be used as a follow expression. We could even remove some of these lookaheads after finishing the CG ro LCG conversion, wheen these lookaheads do not contribute to preventing non-determinism.

First-set and follow-set extraction for CFGs is well known and has been described in the book *Compilers: principles, techniques, & tools* amongst other places [19]. The process remains largely the same for CGs, where regular expressions take the place of terminals, but there are some slight differences. Note that we assume the grammar to contain no more inductively defined production components, and instead only consists of reference and regular expression production components. This will be the case after regular expression conversion has finished. When trying to get the first expression of a sequence of production components, we have to consider that a regular expression may both be able to match the empty string, and a non-empty string. This makes null-checking more complex than with the standard first-set calculation approach. TCNFAs can be used to check whether a regular expression accepts an empty string, and our previously defined function can be used to extract the part of the expression that matches only non-empty strings. The part of the regular expression describing empty matches could be discarded. It may however describe a restriction on what may follow or precede it, in which case discarding it would not accurately reflect the original language. This is not a problem however, since discarding restrictions will only broaden the language. This might make our follow-set less capable of preventing non-determinism, but it wont change the described language. Lastly, the standard first and follow-sets use special symbol $ to represent the end of the input. We do not have to resort to a special symbol, but can instead make use of the regular expression $/(\not>.)/$, which describes no input may follow.

This approach of automatically calculating the allowed lookaheads is effective, but often leads to very complex expressions. This can slow down the conversion procedure significantly. When considering our initial example again, we see that making the choice deterministic, only requires us to prevent *for* being followed by an alphabetic character. The regular expression $/for\not>[a-z]/$ also achieves this and includes all phrases of the original language. Because this expression defines a broader language than the automatically generated one, it has less power to prevent non-determinism. Therefore this should be considered

a choice of the grammar developer. If the grammar developer knows this is the only or most common issue, an automatic transformation can add these negative lookaheads to all keywords. This can be done automatically, by detecting regular expressions accepting alphabetic characters, and adding negative lookaheads for alphabetic characters to them. The calculated follow-set needs to be used when employing this automated approach too. The negative lookbehind might exclude phrases that were originally allowed, thus these entries from the follow-set have to be combined with the negative lookahead. For instance if the specific keyword *each* is allowed directly after *for*, we might end up with the following expression: $/for((\not>[a-z])+(>each))/$. Besides generating lookaheads, the developer could also manually add positive or negative lookaheads to the grammar by hand. These options are more involved and less accurate, so if LG derivation time is of no concern, the fully automatic approach can be employed instead.

5 Implementation

The concepts described in previous chapters are the product of experimentation in Rascal. As such, there is a full implementation of all discussed concepts. This implementation can be referenced for additional details that might be lacking from the report. It also allows us to test the described conversion pipeline, in order to validate the approach and answer our main research question, see Chapter 6. A repository containing this implementation can be found on Github at www.github.com/TarVK/syntax-highlighter.

Figure 7 visualizes how various Rascal files relate to the conversion pipeline architecture. These files consist of one primary function, reflected by the file name, possibly combined with some smaller helper functions. The main entry *Main.rsc* contains a single example conversion that makes use of the *convertGrammar* function.

To support this main conversion pipeline, the *regex* directory includes a standalone regular expression implementation with lookaround, subtraction, and tag support. This implementation does however not include any proper way of using regular expressions for text matching or searches. It instead includes only analytical tools. These tools include a conversion function to obtain TCNFAs from regular expressions, TCNFA minimization and normalization functions, TCNFA combinator functions such as a product function, TCNFA ambiguity detection, and a TCNFA language emptiness check.

In order to get a general impression of the scale and nature of the code, Table 1 specifies the number of lines of code in each of the primary source code directories and Fragment 4 shows part of the implementation for the shape conversion step. This code fragment is contained within the larger shape conversion loop, and illustrated how the closing symbols are generated.

directory	number of LOC	includes
conversion	2653	all mandatory steps to convert a Rascal grammar to a LCG
mapping	1263	all code to map a LCG to a TextMate, Ace, Monarch, or Pygments grammar
determinism	765	all code for determining whether a LCG is deterministic, and lookahead generation functions to improve determinism
specTransformations	202	functions to transform specification grammars, for instance to automatically assign *keyword* scopes to literals
regex	2026	all code related to regular expression analysis
testing	2353	all Rascal code for manual and automated testing
util	18	small utility functions

Table 1: Code Distribution by Directory

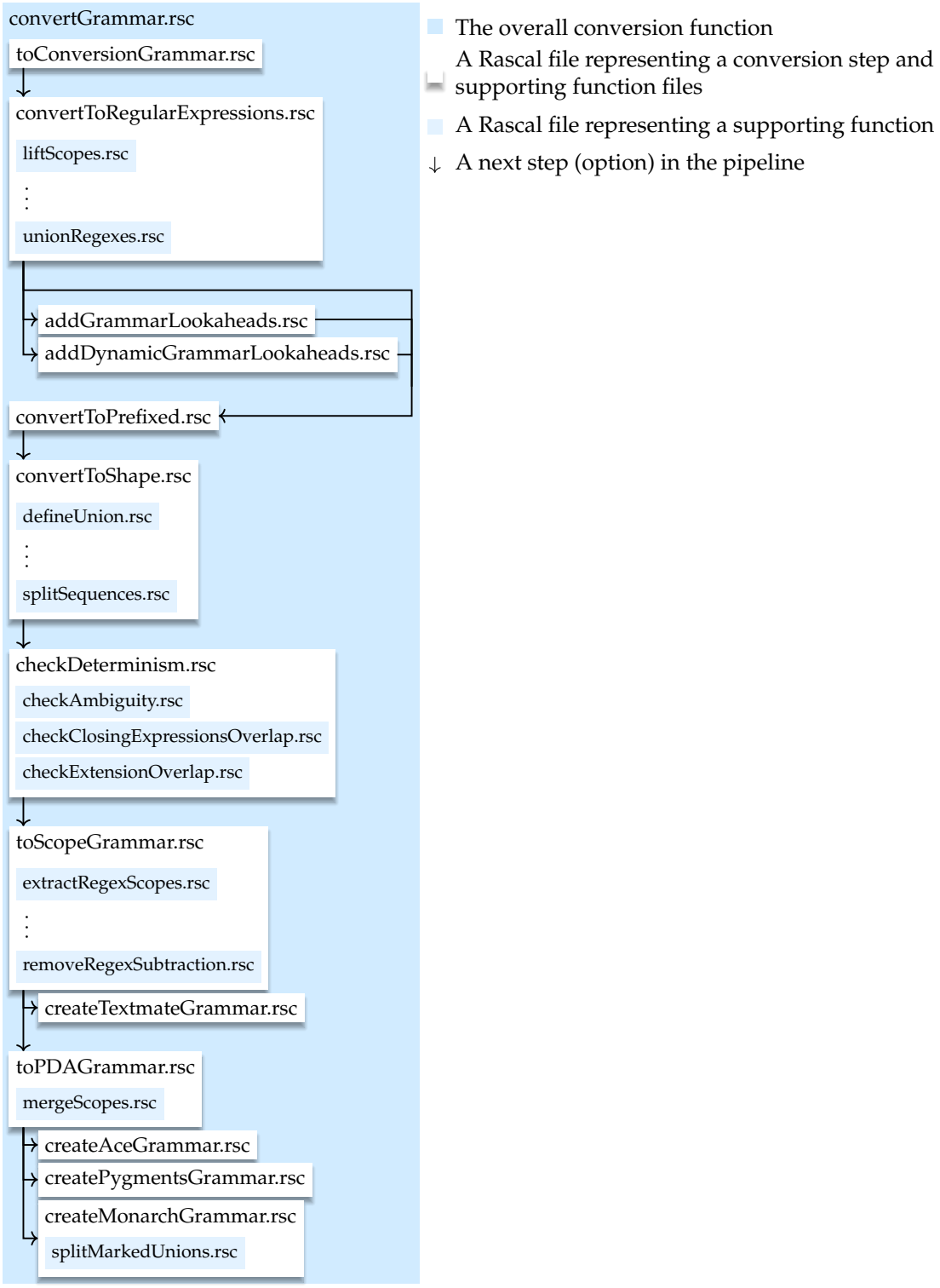

Figure 7: Conversion Pipeline Implementation

```
// Check if there are any new closings left to be defined
definedSymbols = grammar.productions<0>;
set[Symbol] toBeDefinedClosings = {s | s:closed(_, _) <- getReachableSymbols(grammar, false) -
↪  definedSymbols};
if(toBeDefinedClosings == {}) {
    testConfig.log(Progress(), "no new symbols to define");
    break;
}

// Define all undefined but referenced closings
testConfig.log(Progress(), "defining <size(toBeDefinedClosings)> closings");
for(closing <- toBeDefinedClosings) {
    <newProds, isAlias>                = defineClosing(closing, grammar);
    testConfig.log(ProgressDetailed(), "defining <size(newProds)> closing productions");
    if(!isAlias) {
        <mWarnings, newProds, grammar> = combineConsecutiveSymbols(newProds, grammar, testConfig);
        newProds                       = removeLeftSelfRecursion(newProds, testConfig.log);
        newProds                       = removeRedundantLookaheads(newProds, true, testConfig.log);
        <oWarnings, newProds, grammar> = combineOverlap(newProds, grammar, testConfig);
        newProds                       = removeRedundantLookaheads(newProds, false, testConfig.log);
        <sWarnings, newProds, grammar> = splitSequences(newProds, grammar, testConfig);
        <cWarnings, newProds, grammar> = carryClosingRegexes(newProds, grammar, testConfig);
        <nWarnings, newProds>          = checkLeftRecursion(newProds, grammar, testConfig.log);
        testConfig.log(ProgressDetailed(), "finished defining closing productions");

        warnings += mWarnings + oWarnings + sWarnings + cWarnings + nWarnings;
    }
    grammar.productions += {<closing, production> | production <- newProds};
}
testConfig.log(Progress(), "defined <size(toBeDefinedClosings)> closings");
```

Fragment 4: Closing Symbol Generation

5.1 Testing and Debugging

All the code of our conversion pipeline has been written for experimentation purposes. Therefore it is important to have proper testing and debugging tools available. In order to test whether the grammars obtained from the conversion pipeline work well, we have to analyze tokenizations obtained from these grammars. We could use the respective tools that each of the grammar formats is intended for, but this would require us to manually inspect the syntax highlightings for given inputs. This is exactly what we did for Ace, Monarch, and Pygments grammars. We however invested additional development time for TextMate grammars, and created an automated approach.

The TypeScript package *vscode-textmate* allows us to run VSCode's implementation of the TextMate tokenizer from within TypeScript. We wrote a simple TypeScript program that allows us to use Node.js to create tokenizations. This program runs from the terminal and produces a tokenization from a TextMate grammar and an input fragment. The code for this is provided in the *highlighterTesting* directory of the repository. Rascal's *ShellExec* module was used to obtain tokenizations from within Rascal code, allowing us to perform further analysis. Using Rascal's *test* syntax, several test cases were created to verify the conversion pipeline functions as intended. These tests use a Rascal grammar to tokenize a fragment, then obtain a corresponding TextMate grammar to tokenize the same fragment with, and finally compare the specification and output tokenizations. We also used this automated tokenization approach for several experiments, for which we were not sure how well the conversion pipeline would operate. These tests and experiments are discussed in the following chapter.

The previous chapter shows that the conversion pipeline is rather involved. This meant that it took quite some experimentation and development before we managed to obtain any TextMate grammar to test with. During this phase of development, we analyzed the intermediate CG themselves by hand.

To aid us in this task, we created a standalone Rascal data visualization tool called Rascal-vis. This tool allows us to interactively explore Rascal values. It also allows values representing grammars or diagrams

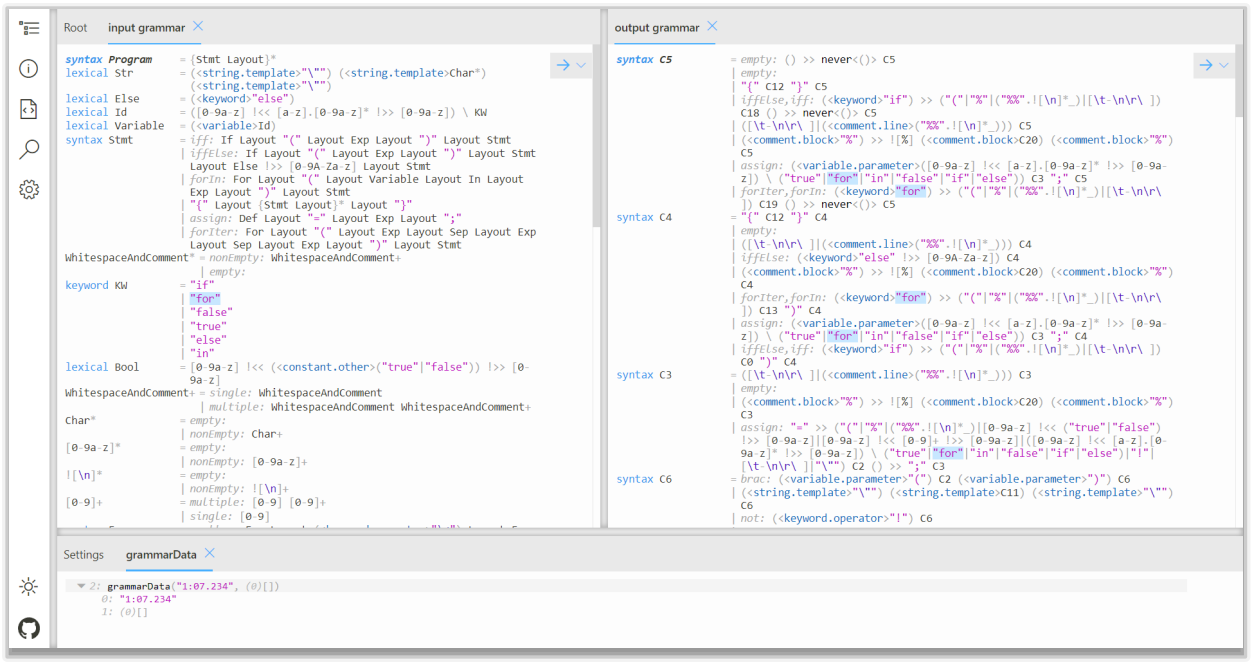

Figure 8: Rascal-vis Grammar Visualization

to be graphically represented, rather than textually. These diagrams are used for visualization of TCNFAs. The tool was developed to be separate from the rest of our project, and as such does not support graphical visualization of CGs. Instead, we convert our CGs to Rascal grammars which are then graphically visualized by our tool. This means that regular expression constructs are represented by Rascal grammar constructs, lookaheads are for instance represented by follow operators. Rascal-vis is designed to run in the web-browser, and can communicate with Rascal code by using Rascal as a server. A value x can be viewed by applying `visualize(x)`, after which x will appear in Rascal-vis to be explored. Rascal-vis includes many features that were valuable during debugging. One of these features is the ability to show multiple panels side by side, and highlight occurrences of a term in all these panels at once. Figure 8 shows an example of how we visualized an input grammar and a CG side by side to look for issues. Rascal-vis also comes with settings and setting profiles, which allows different panel arrangements to be saved per profile. Figure 9 shows how a regular expression and its corresponding TCNFA could be quickly analyzed using the Rascal code shown in Fragment 5. This tool can be found on Github at www.github.com/TarVK/rascal-vis, and may continue to be developed in the future.

```
void main(){
    regexText = "([a-z]*\>[A-Z]*!\>.)((\<tag\>test)|(\<tag\>TEST)|REGEX)";
    r = getCachedRegex(parseRegexReduced(regexText));
    visualizeGrammars(<regexText,r>);
}
```

Fragment 5: Regex Visualization Code

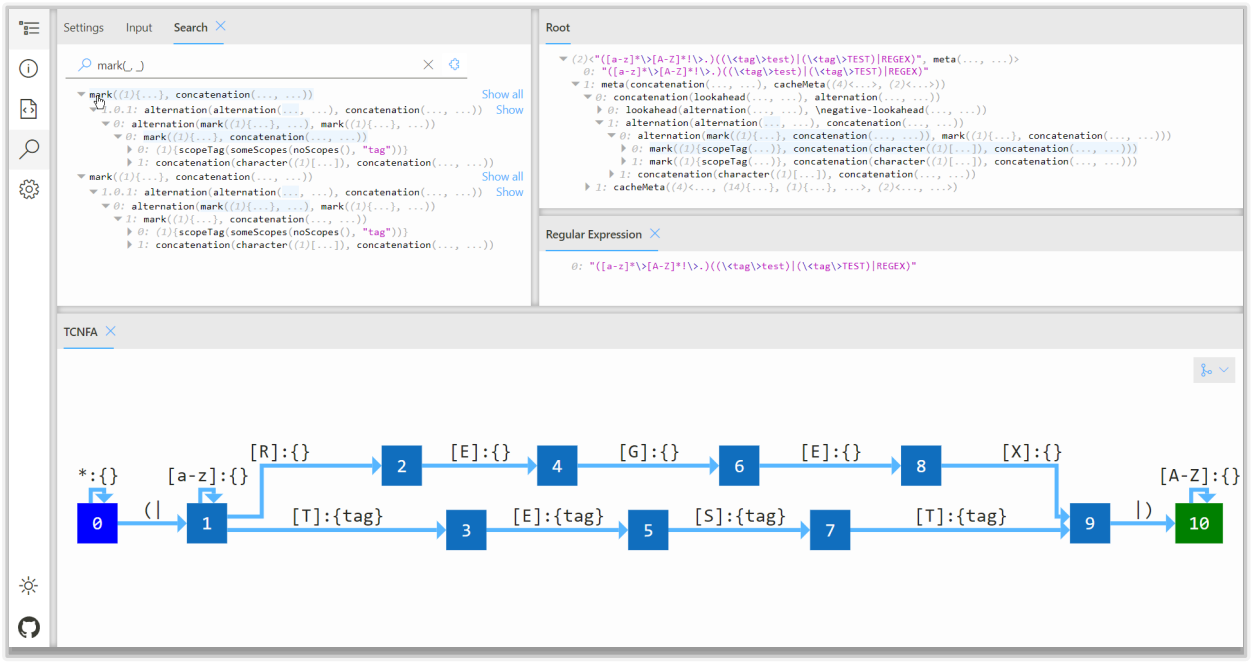

Figure 9: Rascal-vis Regular Expression

6 Testing and Emperical Evaluation

The previous chapter described an approach to obtain a LG for a specification CFG. This approach is believed to obtain a valid LG for any specification grammar, but is not guaranteed to fully retain the specified tokenizations for all input sentences. The next step is to determine how well this approach actually works.

In order to evaluate the quality of the approach, we first have to decide what exactly we are interested in. Several related but distinct attributes of the conversion approach can be identified. The conversion process itself is an algorithm and therefore the standard algorithmic measures of runtime and memory usage apply. The latter is not a major concern for our problem, and will be skipped. Runtime is also not massively important, given that this algorithm only has to be run when a new syntactic language feature for a developed language is released. Given the nature of the algorithm, especially in its current state, changes to the specification might have to be made in order to obtain a specification whose tokenizations are correctly maintained. A slow conversion time will hurt the ability of language developers to iterate on their specifications. Nevertheless, we consider conversion speed of secondary importance, which has not been considered too much in our initial implementation. The quality of the LGs derived by the algorithm has been our primary concern.

The quality of a LGs can be measured in multiple ways once again. We identify 3 main aspects that are of importance:

- Accuracy: do tokenizations correspond to the specification?
- Robustness: how well is tokenization preserved if parts of the document are syntactically incorrect?
- Generalizability: how well are syntactically incorrect parts of the document tokenized?

Accuracy describes how well tokenizations of a LG correspond to those of a specification CFG. If the LG is fully accurate, all possible input phrases have an identical tokenization according to the LG and the specification CFG. Robustness describes how well tokenization is preserved for phrases that are not part of the language of the specification CFG. In this case phrases are considered for which the specification CFG does not specify any tokenization. In order to assess the quality of tokenizations, every considered phrase w has to be matched with a phrase w' within the specification language. This can be done by taking a w' with a minimal distance to w under some distance-measure, such as the Levenshtein distance. The specification tokenization of w' could then be compared to the tokenization of w using the LG. For robustness, only the matching characters between w and w' are considered. Generalizability is then the logical counterpart of robustness, where the non-matching characters of w are considered. In this case, there is no obvious character in w' to compare the scopes to. What the expected tokenization is, is therefore especially unclear and hard to define. one could decide to choose to use a special *error* category for all these characters.

Defining exact measures for these different grammar aspects is not trivial. Especially robustness and generalization get rather complex quite quickly. Our analysis will be limited to the accuracy measure, which also has been the primary focus during the development of the transformation pipeline. More specifically, we define a precision metric for this. We will first define this precision for a given input phrase w and a given LG G with respect to a specification CFG G^*. We would like the precision $\mathcal{P}(w, G, G^*)$ to describe the fraction of correctly tokenized characters in the input, such that 1 describes full precision, and 0 describes the worst possible precision. In this process, we consider scopes as atomic entities being used as tokens, and do not consider individual categories and partial correctness of scopes. Moreover, if a word in the input phrase should have been tokenized as an *identifier* but is not, we would like the length of the identifier to be irrelevant for the precision. Longer identifiers in the input would otherwise result in a worse precision, which does not properly reflect the grammar itself being worse. Otherwise our measure might implicitly describe a bias for accurately tokenizing inputs with long identifiers, compared to short identifiers, since incorrectly tokenizing long identifiers harms the precision more. In order to deal with this, we will group consecutive characters, based on whether they received both identical LG and specification CFG tokens. Let $\mathcal{T}(w, G) \subseteq (T^*)^*$ and $\mathcal{T}(w, G^*) \subseteq (T^*)^*$ define the tokenizations of w for grammar G and G^* respectively.

Then the grouping function $\mathcal{C}$ can be defined as:

$$\begin{aligned}\mathcal{C}(w, G, G^*) = \{&(i, \mathcal{T}(w, G)[i], \mathcal{T}(w, G^*)[i]) \\ &\mid i \in [1..|w|] \\ &\wedge (i == 1 \vee \mathcal{T}(w, G)[i-1] \neq \mathcal{T}(w, G)[i] \vee \mathcal{T}(w, G^*)[i-1] \neq \mathcal{T}(w, G^*)[i])\}\end{aligned}$$

These groups then describe all indices at which a new tokenization pair starts. These can then be used to describe the precision as follows:

$$\mathcal{P}(w, G, G^*) = \frac{|\{i \mid (i, t, t^*) \in \mathcal{C}(w, G, G^*) \wedge t \neq t^*\}|}{|\mathcal{C}(w, G, G^*)|}$$

The precision measures the fraction of groups that got an identical scope assigned from the LG and the specification CFG. The precision of the whole grammar could then be described as the average precision of all phrases in the language.

Determining accuracy, robustness, and generalizability for existing CFGs is outside the scope of this research. Instead we will look at several simple test cases, to support the arguments presented in the previous chapter. These tests will not be able to prove that everything indeed behaves as argued, but at least provides some evidence towards this. Additionally we will perform some more complex experiments to determine the extend of the capabilities and limitations of the described approach. Most of these test cases and experiments focus on the precision of a given input or set of inputs. Some will also focus on the conversion process itself. For all these conversions, except if specified otherwise, negative regex lookaheads are generated as described in Chapter 4.6.

6.1 Regex Subtraction Removal Test

In Section 4.5.1 we described a process for removing the subtraction operator from regular expressions, and claimed that despite this approach not always working, it would work for the most common situations. We will provide tests confirming that it indeed works in the described common cases. Additionally we show that it will indeed fail when there are no anchor expressions to provide context for the subtraction expression.

6.1.1 Setup

We define four Rascal grammars, one corresponding to each non-terminal specified in Grammar 5. These grammars are converted to TextMate grammars, which are then used to tokenize the inputs shown in Table 2. This table lists a name for the test case, the start symbol used of the grammar, the input text for the test case, and whether we expect the textmate grammar to highlight according to the specification. This table provides syntax highlighting per input according to the specification tokenization, where each token type receives a unique color. Table 3 describes the errors we expect in the grammar conversion process itself.

Rascal's grammar syntax largely matches the syntax of our formalisms, but is not identical. The Rascal $A \backslash B$ operator represents keyword reservation, and corresponds to the $\mathit{except}(A, B)$ production symbol of our conversion grammars. Similarly $A >> B$, $A! >> B$, $A << B$, and $A! << B$ represent $\mathit{follow}(A, B)$, $\mathit{not\text{-}follow}(A, B)$, $\mathit{precede}(A, B)$, and $\mathit{not\text{-}precede}(A, B)$ respectively. Moreover, alternative productions are indicated by the vertical bar — in this format, and *@category* annotations label all elements in the production to belong to the specified category. Square brackets groups are used to specify character classes using character ranges, while quoted character sequence represent literal sequence matches. Finally, Rascal uses vertical bars, asterisk characters, and plus characters to represent regular operations similar to in our regular expression formalism.

```
syntax IdentifierA = @category="identifier" [a-z] !<< [a-z]+ !>> [a-z] \ "word"
                   | @category="keyword" "word";

syntax IdentifierB = @category="identifier" [a-z] !<< [a-z]+ \ "word" !>> [a-z]
                   | @category="keyword" "word";

syntax IdentifierC = @category="identifier" [a-z] !<< [a-z]+ \ "word"
                   | @category="keyword" "word";

syntax IdentifierD = @category="identifier" ([a-z]+ \ "word")
                   | @category="keyword" "word";
```

Grammar 5: Identifiers

test-name	grammar	text	expected tokenization	passed
id1A	IdentifierA	something	match	✔
id2A	IdentifierA	aword	match	✔
id3A	IdentifierA	words	match	✔
keywordA	IdentifierA	word	match	✔
id1B	IdentifierB	something	match	✔
id2B	IdentifierB	aword	match	✔
id3B	IdentifierB	words	match	✔
keywordB	IdentifierB	word	match	✔
id1C	IdentifierC	something	match	✔
id2C	IdentifierC	aword	match	✔
id3C	IdentifierC	words	match	✔
keywordC	IdentifierC	word	match	✔
id1D	IdentifierD	something	match	✔
id2D	IdentifierD	aword	match	✔
id3D	IdentifierD	words	mismatch	✔
keywordD	IdentifierD	word	match	✔

Table 2: Identifier Tests

grammar	expected conversion errors	passed
IdentifierA	None	✔
IdentifierB	None	✔
IdentifierC	None	✔
IdentifierD	UnresolvableSubtraction	✔

Table 3: Identifier Conversion Tests

6.1.2 Results

Tables 2 and 3 include the results of the tests, showing that all tests passed. As discussed *IdentifierD* can not be converted while ensuring correctness, as illustrated by the generated error. Accordingly, test *id3D* illustrates that the tokenization is not preserved, because the negative lookahead excluded more phrases than it should have. Table 4 shows the specified tokenization and the tokenization received from the TextMate grammar.

To get a better understanding for the exact output of our algorithm, the TextMate grammar generated for *IdentifierA* is illustrated in Grammar 6. Note that this grammar includes some redundant artifacts like the *empty* pattern that does not match any inputs. These patterns are generated by the conversion algorithm, but could easily be removed yielding a grammar that tokenizes equivalently. Table 5 shows the JSON representation of tokenizations for several test-cases. Each tokenization is a list of scopes – one scope per character – where each scope is a list of categories.

test-name	specified tokenization	output tokenization
id3D	`words`	`words`

Table 4: Identifier Mismatching Tokenizations

```
{
    "name": "test",
    "scopeName": "source.test",
    "patterns": [{"include": "#C1"}],
    "repository": {
        "C1": {
            "patterns": [
                {"include": "#empty"},
                {"include": "#S"}
            ]
        },
        "S": {
            "begin": "(?:(word)|(?!(?<![a-z])(?<![a-z])word(?![a-z]))((?<![a-z])
↪   [a-z]+(?![a-z])))(?=(?!.))",
            "end": "(?:x(?<!x))",
            "beginCaptures": {
                "1": {"name": "keyword"},
                "2": {"name": "identifier"}
            },
            "endCaptures": [],
            "patterns": [{"include": "#C0"}]
        },
        "C0": {"include": "#empty"},
        "empty": {
            "match": "(?:x(?<!x))",
            "captures": []
        }
    }
}
```

Grammar 6: IdentifierA TextMate

test-name	type	raw tokens
id2A	specification	`[["identifier"],["identifier"],["identifier"],["identifier"],["identifier"]]`
keywordC	specification	`[["keyword"],["keyword"],["keyword"],["keyword"]]`
id3D	specification	`[["identifier"],["identifier"],["identifier"],["identifier"],["identifier"]]`
id3D	output	`[[],["identifier"],["identifier"],["identifier"],["identifier"]]`

Table 5: Identifier Raw Tokens

6.2 Overlap Combining Test

We argued that overlapping production combining is a critical step for dealing with non-determinism. We will compare how a grammar with or without such a step will behave.

6.2.1 Setup

We define a Rascal grammar *Merge* in Grammar 7. This grammar is converted to two TextMate grammars – one skipping the production combining step – which are then used to tokenize the inputs shown in Table 6. This table is similar to Table 2, but also indicates whether the overlap combining has been skipped.

Note that the expected behavior for several tests, such as *alt1Disabled*, is unknown. Either *alt1Disabled* or *alt2Disabled* should tokenize according to specification, but it is unknown which of these inputs tokenizes correctly. This is the case because an arbitrary production order is generated when converting the LCG to a scoped grammar. These unknown test cases will pass regardless of their result, they are merely listed to explicitly state the behavior to be fundamentally unknown.

```
syntax Merges = Merge*;
syntax Merge  = A SB D
              | A SC D;
syntax SB     = "(" SB ")" | B;
syntax SC     = "(" SC ")" | C;
lexical A     = @category="1" "a";
lexical B     = @category="2" "b";
lexical C     = @category="3" "c";
lexical D     = @category="4" "d";

layout Layout = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 7: Overlap Combining

6.2.2 Results

All tests passed, as illustrated in Table 6. Table 7 shows the specified tokenization and the tokenization received from the TextMate grammar for test cases where no match is expected. Grammars 8 and 9 show the TextMate grammars generated from Grammar 7 with overlap combining enabled and disabled respectively.

test-name	grammar	overlap combining	text	expected tokenization	passed
alt1	Merges	enabled	abd	match	✔
alt2	Merges	enabled	acd	match	✔
altBoth	Merges	enabled	abdacd	match	✔
alt1grouped	Merges	enabled	a(b)d	match	✔
alt2grouped	Merges	enabled	a(c)d	match	✔
altBothGrouped	Merges	enabled	a(b)da(c)d	match	✔
alt1Disabled	Merges	disabled	abd	unknown	✔
alt2Disabled	Merges	disabled	acd	unknown	✔
altBothDisabled	Merges	disabled	abdacd	mismatch	✔
alt1groupedDisabled	Merges	disabled	a(b)d	unknown	✔
alt2groupedDisabled	Merges	disabled	a(c)d	unknown	✔
altBothGroupedDisabled	Merges	disabled	a(b)da(c)d	mismatch	✔

Table 6: Overlap Combining Tests

test-name	specified tokenization	output tokenization
alt1Disabled	abd	abd
alt2Disabled	acd	acd
altBothDisabled	abdacd	abdacd
alt1GroupedDisabled	a(b)d	a(b)d
alt2GroupedDisabled	a(c)d	a(c)d
altBothGroupedDisabled	a(b)da(c)d	a(b)da(c)d

Table 7: Overlap Combining Mismatching Tokenizations

```
{
    "name": "test",
    "scopeName": "source.test",
    "patterns": [{"include": "#C0"}],
    "repository": {
        "C0": {
            "patterns": [
                {"include": "#empty"},
                {"include": "#single,multiple"},
                {"include": "#S"}
            ]
        },
        "C1": {
            "patterns": [
                {"include": "#T"},
                {"include": "#single,multiple"},
                {"include": "#T1"},
                {"include": "#S1"}
            ]
        },
        "S": {
            "begin": "(a)",
            "end": "(d)",
            "beginCaptures": {"1": {"name": "1"}},
            "endCaptures": {"1": {"name": "4"}},
            "patterns": [{"include": "#C1"}]
        },
        "T1": {
            "match": "(c)",
            "captures": {"1": {"name": "3"}}
        },
        "S1": {
            "begin": "\\(",
            "end": "\\)",
            "patterns": [{"include": "#C1"}]
        },
        "empty": {"match": "(?:x(?<!x))"},
        "T": {
            "match": "(b)",
            "captures": {"1": {"name": "2"}}
        },
        "single,multiple": {"match": "[\\t-\\n\\r ]"}
    }
}
```

Grammar 8: Merge TextMate

```
{
    "name": "test",
    "scopeName": "source.test",
    "patterns": [{"include": "#C2"}],
    "repository": {
        "C2": {
            "patterns": [
                {"include": "#empty"},
                {"include": "#single,multiple"},
                {"include": "#S1"},
                {"include": "#S2"}
            ]
        },
        "C0": {
            "patterns": [
                {"include": "#T"},
                {"include": "#single,multiple"},
                {"include": "#S"}
            ]
        },
        "C1": {
            "patterns": [
                {"include": "#single,multiple"},
                {"include": "#T1"},
                {"include": "#S3"}
            ]
        },
        "S1": {
            "begin": "(a)",
            "end": "(d)",
            "beginCaptures": {"1": {"name": "1"}},
            "endCaptures": {"1": {"name": "4"}},
            "patterns": [{"include": "#C0"}]
        },
        "S": {"begin": "\\(", "end": "\\)", "patterns": [{"include": "#C0"}]},
        "T": {"match": "(c)", "captures": {"1": {"name": "3"}}},
        "S2": {
            "begin": "(a)",
            "end": "(d)",
            "beginCaptures": {"1": {"name": "1"}},
            "endCaptures": {"1": {"name": "4"}},
            "patterns": [{"include": "#C1"}]
        },
        "S3": {"begin": "\\(", "end": "\\)", "patterns": [{"include": "#C1"}]},
        "T1": {"match": "(b)", "captures": {"1": {"name": "2"}}},
        "empty": {"match": "(?:x(?<!x))"},
        "single,multiple": {"match": "[\\t-\\n\\r ]"}
    }
}
```

Grammar 9: Merge Without Overlap Combining TextMate

6.3 Strict Overlap Combining Test

When we introduced the overlap combining transformation, we mentioned how we can relax the structure when combining productions. We made the lookahead of the inner sequences optional, increasing the risk of non-determinism in the final grammar. Here we will provide several tests to illustrate that in simple cases the relaxed grammar works equally well, while being simpler to compute.

6.3.1 Setup

We define a Rascal grammar *Merge2* in Grammar 10. This grammar is converted to two TextMate grammars – one with strict overlap combining and one without– which are then used to tokenize the inputs shown in Table 8. This table is similar to Table 2, but also indicates whether strict or relaxed overlap combining has been used. Note that the default mode used in other tests is the relaxed mode.

Besides checking the tokenizations according to specification, we also perform tests on input phrases that are syntactically incorrect. For these phrases, we manually specified the expected tokenizations, provided in Table 9 in the form of syntax highlighting. These test cases illustrate that the behavior of the resulting grammars is not identical, despite both fully retaining the specified tokenizations. Finally, we check whether strict overlap merging requires more iterations of the shape conversion loop.

```
syntax Merge2 = A Merge2
              | A Merge2 B Merge2
              | C;
lexical A     = @category="1"  "a";
lexical B     = @category="2"  "b";
lexical C     = @category="3"  "c";

layout Layout = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 10: Overlap Combining Of Optional Suffix

test-name	grammar	combining mode	text	expected tokenization	passed
noSuffix	Merge2	relaxed	a c	match	✔
1suffix	Merge2	relaxed	a c b c	match	✔
2suffix	Merge2	relaxed	a a c b c b c	match	✔
2suffixSeq	Merge2	relaxed	a c b a c b c	match	✔
noSuffixStrict	Merge2	strict	a c	match	✔
1suffixStrict	Merge2	strict	a c b c	match	✔
2suffixStrict	Merge2	strict	a a c b c b c	match	✔
2suffixSeqStrict	Merge2	strict	a c b a c b c	match	✔

Table 8: Overlap Combining Of Optional Suffix Tests

6.3.2 Results

All tests passed, as illustrated in Tables 8 and 9. Additionally, converting the grammar using the strict overlap combining mode required 3-iterations, while the relaxed version required 2-iterations and was thus indeed slightly simpler. The difference in tokenization behavior is illustrated in tests *tripleSuffix* and

test-name	grammar	combining mode	text	expected tokenization	passed
onlySuffix	Merge2	relaxed	`b c`	match	✔
doubleSuffix	Merge2	relaxed	`a c b c b c`	match	✔
tripleSuffix	Merge2	relaxed	`a c a c b c b c b c`	match	✔
onlySuffixStrict	Merge2	strict	`b c`	match	✔
doubleSuffixStrict	Merge2	strict	`a c b c b c`	match	✔
tripleSuffixStrict	Merge2	strict	`a c a c b c b c b c`	match	✔

Table 9: Overlap Combining of Optional Suffix Generalization Tests

tripleSuffixStrict. We see that the strict version can correctly identify that only a single *b* may follow here, and that the second one is syntactically incorrect. The relaxed version does not exhibit such behavior.

6.4 Infinite Recursion Test

When we discussed the conversion pipeline, we noted that optional suffixes in productions could lead to the conversion process never terminating and instead generating increasingly complex symbols. We then provided a simple patch by detecting this type of recursion, and resolving it by relaxing such symbols. The previous test dealt with a grammar with an optional suffix. here we verify that our patch is indeed required.

6.4.1 Setup

We try to convert Grammar 10 to a TextMate grammar, while disabling the recursion detection mechanism. Additionally, we enable strict overlap combining mode to force the grammar to be strict enough for this to occur. We assume that this will lead to indefinite recursion, but this is not directly detectable. Instead we assume this process will indeed iterate indefinitely, if it does not terminate after 16 iterations. Note that the original conversion only required 3 iterations before the recursive structure detection kicked in.

6.4.2 Results

The process does indeed not terminate before 16 iterations are reached. However, if strict mode is not enabled the process terminates after only 2 iterations. We were not able to create a test case that iterates indefinitely for the relaxed overlap combining mode.

6.4.3 Conclusion

We are not able to automatically verify that disabling the recursion check results in non-termination, but have shown potential of an unreasonably large number of iterations occurring under the right circumstances. Therefore it is safer to keep this check enabled by default.

6.5 Exponential Merging Test

When productions exist with optional suffixes, these suffixes only become accessible when the mandatory prefix has been encountered. This is shown in the *onlySuffix* test case provided in the strict overlap combining test. To achieve this, the grammar will encode what prefixes have already been encountered in the form of a non-terminal. This means that if there are n recursive sequences with optional suffixes, there are 2^n possible combinations of what suffixes have already been encountered, and thus at least 2^n corresponding non-terminals. We will verify whether this is indeed the case based on a test.

6.5.1 Setup

We provide Grammar 11 with three different right-recursive productions with optional suffixes. The optionality of suffixes is created by providing alternative productions that include the suffix as a mandatory component. Overlap combining will merge these productions together, leading to a non-terminal symbol which encodes optionality of the the suffix. We will analyze the output LCG and verify that for each possible combination of allowed and disallowed suffixes, a corresponding non-terminal symbol is present. This is done by checking for productions starting with a regular expression that accept the given character. Hence for every $S \subset b, d, f$, we expect a single non-terminal symbol to exist with a production starting with a regular expression accepting s for $s \in S$, and that does not have a single production starting with a regular expression accepting e for $e \in b, d, f - S$.

```
syntax Merge3 = A A Merge3
              | A A Merge3 B Merge3
              | C C Merge3
              | C C Merge3 D Merge3
              | E E Merge3
              | E E Merge3 F Merge3
              | G;
lexical A     = @category="1"  "a";
lexical B     = @category="2"  "b";
lexical C     = @category="3"  "c";
lexical D     = @category="4"  "d";
lexical E     = @category="5"  "e";
lexical F     = @category="6"  "f";
lexical G     = @category="7"  "g";

layout Layout = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 11: Multiple Optional Suffixes

6.5.2 Results

The test passes as expected. The double non-terminals provided as prefixes in the grammar are necessary to get conversion to behave exactly like this however. If only single non-terminals were used here, the regex conversion would merge all productions without a prefix together into a single production. As a result, all suffixes would also be enabled whenever one of these prefixes match. We had to add some extra complexity to the productions to prevent this from happening. This also accurately reflects the more complex nature of productions you would encounter in grammars of real languages. This demonstrates how the size of the output grammar could easily become exponential depending on the exact structure of the input grammar.

6.6 Determinism Checks Test

Our conversion pipeline has a dedicated step to check whether the resulting grammar is deterministic. This step does not affect the output grammar itself, but can warn about potential issues. We will provide some test cases that illustrate how problematic cases are caught by this step.

6.6.1 Setup

We define a Rascal grammar for each of the possible non-deterministic behaviors: grammar ambiguity, regex extensibility, optional closing. These three grammars are provided together in Grammar 12. We also provide slightly modified versions of each of these grammars which fully retain their tokenizations in Grammar 13. Once again, these grammars are converted to TextMate grammars, which are then used to tokenize the inputs shown in Table 10. This table again lists the test case name, the grammar start symbol, input text and specification tokenization, and whether the TextMate tokenization should be identical. Note that for the *extension* test, whether the tokenization would match or not is unknown. This is the case, because it depends on the ordering of productions, which happens arbitrarily. As such, this test will pass regardless of outcome. It is merely listed to explicitly state the behavior to be fundamentally unknown. Finally, Table 11 describes the errors we expect in the grammar conversion process itself.

```
syntax Ambiguity    = AmbiguityAlt*;
syntax AmbiguityAlt = KW1 "!" | KW2 "?";
lexical KW1         = @category="1" "s";
lexical KW2         = @category="2" "s";

syntax Extensions   = Extension*;
syntax Extension    = @category="1" "a"
                    | @category="2" "s"
                    | @category="3" "as";

syntax Closings     = Closing*;
syntax Closing      = "(" Closing ")"
                    | @category="1" ")";

layout Layout       = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 12: Non-determinism

```
syntax NoAmbiguity     = NoAmbiguityAlt*;
lexical NoAmbiguityAlt = KW1 "!" | KW2 "?";
lexical KW1            = @category="1" "s";
lexical KW2            = @category="2" "s";

syntax NoExtensions    = NoExtension*;
syntax NoExtension     = @category="1" "a" !>> "s"
                       | @category="2" "s"
                       | @category="3" "as";

syntax NoClosings      = NoClosing*;
syntax NoClosing       = "(" NoClosing ")"
                       | @category="1" "s" ")";

layout Layout          = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 13: Determinism

test-name	grammar	text	expected tokenization	passed
ambiguity	Ambiguity	`s ! s ?`	mismatch	✔
noAmbiguity	NoAmbiguity	`s! s?`	match	✔
extension	Extension	`as`	unknown	✔
noExtension	NoExtension	`as`	match	✔
closing	Closing	`())`	mismatch	✔
noClosing	NoClosing	`(s)`	match	✔

Table 10: Determinism Tests

grammar	expected conversion errors	passed
Ambiguity	Ambiguity	✔
NoAmbiguity	None	✔
Extension	ExtensionOverlap	✔
NoExtension	None	✔
Closing	ClosingOverlap	✔
NoClosing	None	✔

Table 11: identifier Conversion Tests

6.6.2 Results

All tests passed, as illustrated in Tables 10 and 11. Table 12 shows the specified tokenization and the tokenization received from the TextMate grammar for test cases where no match is expected.

test-name	specified tokenization	output tokenization
ambiguity	`s ! s ?`	`s ! s ?`
extension	`as`	`as`
closing	`())`	`())`

Table 12: Non-Determinism Mismatching Tokenizations

6.6.3 Analysis

The reason that *Ambiguity* causes ambiguity *NoAmbiguity* does not, lies in one of the limitations of syntax highlighters. As previously discussed, syntax highlighters perform regex searches only on a single line. Because of this the exclamation mark and question mark can not be combined into a single regular expression if layout is allowed between them. Rascal automatically interweaves the *Layout* symbol into any productions of non-terminals declared as *syntax*. For *NoAmbiguity*, we instead declared the non-terminal as *lexical* which prevents Rascal from interweaving the layout.

Our *Extension* grammar happens to be ambiguous, but similar choices can come up in more complex grammars that are not ambiguous. Here we solved the problem by removing the ambiguity using a negative follow declaration. Finally, because any non-terminal can occur zero or more times in the output grammar, the nested occurrence of *Closing* can also occur zero or more times.

Because of this, when a closing bracket is encountered, the tokenizer can not deterministically decide whether it comes from the first or second production, and hence causes closing non-determinism.

6.7 Overlap Non-Determinism Experiment

The overlap combining technique we described is fairly simple. We know this technique has its limitations, and we expect these limitations cause issues for constructs encountered in grammars of existing languages. In this experiment we attempt to create a situation where this indeed causes an issue, and see whether a manual fix to the grammar resolves this issue.

6.7.1 Setup

We define Rascal Grammar 14 with two productions with a common prefix but without a common suffix. Additionally we define alternative Grammar 15 that specifies the same language and tokenizations. We convert both grammars to TextMate grammars and provide some input cases to test the grammars on, shown in Table 13. Note that the text followed by colons in these grammars, merely represent labels for

the corresponding productions and do not affect tokenization. The first production of *Exp1* is for instance labeled *group*.

```
syntax Program1      = Exp1*
syntax Exp1          = group:  "(" Exp1 ")"
                     | lambda: "(" {Variable ","}* ")" Lambda Exp1
                     | val:    Variable;

lexical Lambda       = @category="keyword" "=\>";
lexical Variable     = @category="variable" Id;
lexical Id           = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]);

layout Layout        = [\ \t\n\r]* !>> [\ \t\n\r%];
```

Grammar 14: Overlap Non-Determinism

```
syntax Program2      = Exp2*
syntax Exp2          = group:  "(" Exp2 ")"
                     | lambda: ("(" {Variable ","}* ")") Lambda Exp2
                     | val:    Variable;

lexical Lambda       = @category="keyword" "=\>";
lexical Variable     = @category="variable" Id;
lexical Id           = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]);

layout Layout        = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 15: Overlap Non-Determinism Fixed

input-name	tokenization grammar	text
function	Program1	`(arg)=>arg`
functionReturnGroup	Program1	`(arg)=>(arg)`
functionInGroup	Program1	`((arg)=>arg)`

Table 13: Overlap Non-Determinism Inputs

6.7.2 Hypothesis

We expect that the conversion of Grammar 14 will generate some errors, due to not being able to balance brackets. During the conversion, only common suffixes can be used as closing expressions to define new scopes, therefore the lack of a common suffix means that the TextMate grammar can not keep count of how many closing brackets can be encountered. This should be reflected in the *functionInGroup* test case.

On the other hand, we expect conversion of Grammar 15 to be perfect. This should be the case because adding brackets around `"(" {Variable ","}* ")"` should lead to a dedicated non-terminal symbol being created by Rascal for this sequence. During the prefix conversion step, this production will be split into two parts: the initial sequence ending in the closing bracket, and the remainder starting at the lambda-arrow.

This initial sequence shares the closing suffix with the group production, and hence conversion should work perfectly.

6.7.3 Results

Both grammars were converted perfectly without any errors being generated. Accordingly, all test inputs obtained a full precision, as described in Table 14.

input-name	tokenization grammar	precision
function	Program1	100%
functionReturnGroup	Program1	100%
functionInGroup	Program1	100%
function	Program2	100%
functionReturnGroup	Program2	100%
functionInGroup	Program2	100%

Table 14: Overlap Non-Determinism Precisions

6.7.4 Discussion

The results were better than expected. After careful consideration, there was a mistake in the our initial Hypothesis. The conversion process does not work better than expected, but instead our assumption of bracket counting being relevant was wrong. The resulting TextMate grammar of Grammar 14 can indeed not keep track of how many more closing brackets are allowed, but this also is not required in order to highlight accurately. This would only be required when the same characters are tokenized differently inside or outside of these bracketed constructs.

Despite making a mistake in our reasoning, the insight provided by this experiment is still valuable. It demonstrates that the conversion pipeline can often get away with creating non-ideal TextMate grammars, as long as the described tokenizations are not massively complex. This also hints at the idea that reducing the level of detail of tokenization in the original specification, can help with successfully converting the grammar.

6.8 Overlap Non-Determinism With Context Experiment

In Experiment 6.7 we learned that overlap merging with distinct suffixes is not always problematic. Here we will attempt to produce a situation where it is problematic, and see whether a simple manual fix to our grammar solves the issue.

6.8.1 Setup

We define Rascal Grammar 16 with two productions with a common prefix but without a common suffix. In this grammar we also add productions for defining a string with expression interpolation. This structure will force some type of bracket counting to be necessary for correct tokenizations. Additionally we define alternative Grammar 17 that specifies the same language and tokenizations. We again convert both grammars to TextMate grammars and provide some input cases to test the grammars on, shown in Table 15.

```
syntax Program1      = Exp1B*
syntax Exp1B         = group:                    "(" Exp1B ")"
                     | lambda:                   "(" {Variable ","}* ")" Lambda Exp1B
                     | @category="string" str:   "\"" Char* "\""
                     | var:                      Variable;

lexical Char         = char:                           ![\\\"$]
                     | dollarChar:                     "$" !>> "("
                     | @category="constant" escape:    "\\"![]
                     | @category="embedded" embedded:  "$(" Layout Exp1B Layout
                     ↪  ")";
lexical Lambda       = @category="keyword" "=\>";
lexical Variable     = @category="variable" Id;
lexical Id           = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]);

layout Layout        = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 16: Overlap Non-Determinism With String

```
syntax Program2      = Exp2B*
syntax Exp2B         = group:                    "(" Exp2B ")"
                     | lambda:                   "(" {Variable ","}* ")" Lambda Exp2B
                     | @category="string" str:   "\"" Char* "\""
                     | var:                      Variable;

lexical Char         = char:                           ![\\\"$]
                     | dollarChar:                     "$" !>> "("
                     | @category="constant" escape:    "\\"![]
                     | @category="embedded" embedded:  "$(" Layout Exp2B Layout
                     ↪  ")";
lexical Lambda       = @category="keyword" "=\>";
lexical Variable     = @category="variable" Id;
lexical Id           = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]);

layout Layout        = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 17: Overlap Non-Determinism With String Fixed

input-name	tokenization grammar	text
function	Program1B	`(arg)=>arg`
functionReturnGroup	Program1B	`(arg)=>(arg)`
functionInGroup	Program1B	`((arg)=>arg)`
variableInString	Program1B	`"text $(var) okay"`
groupedVariableInString	Program1B	`"text $((var)) okay"`
functionInString	Program1B	`"text $((arg)=>arg) okay"`

Table 15: Overlap Non-Determinism With String Inputs

6.8.2 Hypothesis

We expect that conversion of Grammar 16 results in an error, due to the created grammar not being able to determine when a string continues after an interpolated expression. As a result, we expect tokenization errors for inputs *groupedVariableInString* and *functionInString*. On the other hand, we expect Grammar 17 to be converted without generating any errors. The same reasoning here applies as for Experiment 6.7, where adding brackets around a part of the production causes the production to be split into 2 parts during the conversion process. This allows the arguments sequence and the group production to be combined without any issues.

6.8.3 Results

Conversion of Grammar 16 yields a closing overlap error, while conversion of Grammar 17 yields no errors. Table 16 shows the precisions of the provided input phrases. Table 17 shows the exact non-perfect tokenizations. Additionally, we find that the entire conversion of Grammar 16 requires 28 seconds, while conversion of Grammar 17 requires only 9 seconds.

input-name	tokenization grammar	precision
function	Program1B	100%
functionReturnGroup	Program1B	100%
functionInGroup	Program1B	100%
variableInString	Program1B	100%
groupedVariableInString	Program1B	83.33% (5 out of 6 groups)
functionInString	Program1B	62.50% (5 out of 8 groups)
function	Program2B	100%
functionReturnGroup	Program2B	100%
functionInGroup	Program2B	100%
variableInString	Program2B	100%
groupedVariableInString	Program2B	100%
functionInString	Program2B	100%

Table 16: Overlap Non-Determinism With String Precisions

input-name	specified tokenization	output tokenization
groupedVariableInString	`"text $((var)) okay"`	`"text $((var)) okay"`
functionInString	`"text $((arg)=>arg) okay"`	`"text $((arg)=>arg) okay"`

Table 17: Overlap Non-Determinism With String Mismatching Tokenizations

6.8.4 Discussion

The lack of bracket counting abilities indeed affects the grammars ability to tokenize properly in this situation. This was correctly identified by the conversion process and warned about. The modification to the grammar that we expected to resolve this issue, indeed resolved the issue. Additionally, we find that grammar complexity decreases by applying this fix. This makes the fix desireable, regardless of whether it affects correctness.

6.9 Type Recognition Experiment

Several languages such as Java and Rascal specify type information before function or variable names. It would be nice to have syntax highlighting that differentiates between types and variables based on this

positional information. This is trivial for primitive types, but becomes more complex when nested type parameters are allowed. We will test whether our conversion pipeline can handle these cases.

6.9.1 Setup

We define two Rascal grammars, Grammar 18 defining assignment syntax with simple types, and Grammar 19 defining assignment syntax with recursive types. We convert both grammars to TextMate grammars and provide some input cases to test the grammars on, shown in Table 18. For the conversion of Grammar 19 positive lookaheads are generated foreach regular expression. In other tests and experiments we generated negative lookaheads of alphabetic characters since these are usually adequate, but here we may need to provide more information in order to prevent non-determinism in the final grammar. For this grammar we generate lookaheads that look two regular expressions ahead, to increase chances of the final grammar being deterministic even further.

Note that in Grammar 19 the layout expression is split into two parts. Together these parts define an equivalent language as the layout present in Grammar 18. We separated these, because lookaheads can not look past newline expressions, but in order to tell variables and types apart we have to look past optional whitespace. We know an identifier must be a variable if it is followed by an equality character. Similarly we know an identifier must be a type if it is followed by another identifier, or an array bracket or type parameter bracket. We know conversion can not be perfect, since it is possible that these lookahead symbols occur on the next line. However by separating the whitespace that allows for newlines and the one that does not, it might be possible to create lookaheads that do work as intended as long as all characters are on the same line.

```
syntax Program3      = Stmt*;
syntax Stmt          = Type Variable "=" Exp ";";

syntax Exp           = \brackets: "(" Exp ")"
                     | var:       Variable;

syntax Type          = \type:                          TypeVariable
                     | @category="primitive" number:   "number"
                     | @category="primitive" string:   "string"
                     | @category="primitive" boolean:  "bool";

lexical Variable     = @category="variable" Id;
lexical TypeVariable = @category="type" Id;

keyword KW           = "bool"|"number"|"string";
lexical Id           = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]) \ KW;

layout Layout        = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 18: Typed Assignment

```
syntax Program4      = Stmt*;
syntax Stmt          = Type Variable "=" Exp ";";

syntax Exp           = \brackets: "(" Exp ")"
                     | var:       Variable;

syntax Type          = \type:                          TypeVariable
                     | ar:                             Type "[]"
                     | apply:                          Type "\<" {Type ","}+ "\>"
                     | @category="primitive" number:   "number"
                     | @category="primitive" string:   "string"
                     | @category="primitive" boolean:  "bool";

lexical Variable     = @category="variable" Id;
lexical TypeVariable = @category="type" Id;

keyword KW           = "bool"|"number"|"string";
lexical Id           = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]) \ KW;

layout Layout        = withoutNewline: [\ \t]* !>> [\ \t\n\r]
                     | newline:        [\ \t]*[\n\r][\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 19: Complex Typed Assignment

input-name	tokenization grammar	text
primitive	Program4	`bool variable = value;`
type	Program4	`mytype variable = value;`
complexType	Program4	`mytype<number[]> variable = value;`

multiline	Program4	`mytype`
		` variable`
		` = value;`

Table 18: Typed Assignment Inputs

6.9.2 Hypothesis

We expect Grammar 18 to convert perfectly, and thus also tokenize perfectly for supported inputs. Since *Type* non-terminal symbol of Grammar 19 can not be converted to a single regular expression, we expect Grammar 19 to generate errors in the conversion process. When *Type* is not converted to a regular expression, the assignment production will be split into two parts during the prefix conversion step of the conversion pipeline. This removes a lot of our structural data from the grammar. But by providing a sufficiently large lookahead in the regular expression before this structure was lost, variables and type should still be differentiable as long as the entire assignment appears on the same line. For this reason we expect Grammar 19 to tokenize all inputs perfectly, except for the multiline input that provides no differentiable lookahead context.

6.9.3 Results

Grammar 18 is converted without any errors, while conversion of Grammar 19 results in several errors. Table 19 shows the precisions of the provided input phrases. Additionally, we find that the entire conversion

of Grammar 18 requires 14 seconds, while conversion of Grammar 19 requires around 29 minutes.

input-name	tokenization grammar	precision
primitive	Program3	100%
type	Program3	100%
primitive	Program4	100%
type	Program4	100%
complexType	Program4	100%
multiline	Program4	100%

Table 19: Typed Assignment Precisions

6.9.4 Discussion

Grammar 18 could easily be converted to an accurate TextMate grammar as expected. Meanwhile converting Grammar 19 to a TextMate grammar results in errors, as was expected. Despite these errors, all inputs were tokenized correctly. This contradicts our hypothesis, which expected the multiline input to be tokenized incorrectly. After manual analysis of the resulting TextMate grammar, we found that the arbitrary production ordering of the grammar happened to work out perfectly by coincidence. In the initial state, type matching takes precedence over variable matching. When this match is performed, the system switches state (in order to now also allow type suffixes), where variable matching happens to take precedence over type matching. After performing more tests, we do find that this coincidental preservation of the structure does not go past a single assignment. Table 20 shows an additional input case with non-perfect tokenization.

grammar	specified tokenization	output tokenization

Program4	`mytype`
	`  variable`
	`  = value;`
	`mytype`
	`  variable`
	`  = value;`
		`mytype`
		`  variable`
		`  = value;`
		`mytype`
		`  variable`
		`  = value;`

Table 20: Typed Assignment Mismatching Tokenizations

6.10 Type Recognition Restructuring Experiment

In Experiment 6.9 we saw that with a sufficiently large lookahead, our typed assignment grammar could tokenize fairly well despite not being perfect. Several sacrifices had to be made to achieve this however. The added regular expression lookahead complexity as well as the non-deterministic nature of the grammar lead to the conversion pipeline requiring 29 minutes to finish, and no correctness guarantees could be given for the resulting TextMate grammar. Instead of relying on lookaheads, restructuring the grammar itself may yield a better tokenizer.

6.10.1 Setup

We define Rascal Grammar 20 defining assignment syntax with simple types. This grammar is a manual rewrite of Grammar 19, specifying the same language using a different structure. We convert this grammar to a TextMate grammar and use this on the inputs shown in Table 18. For this conversion, we use the default generated negative lookaheads that are also used elsewhere.

```
syntax Program4B     = Stmt*;
syntax Stmt          = TypeWord TypeSuffix Variable "=" Exp ";";

syntax Exp           = bracketss: "(" Exp ")"
                     | var:       Variable;

syntax Type          = TypeWord TypeSuffix;
syntax TypeSuffix    = ar:    "[]" TypeSuffix
                     | apply: "\<" {Type ","}+ "\>" TypeSuffix
                     | ;
syntax TypeWord      = \type:                          TypeVariable
                     | @category="primitive" number:  "number"
                     | @category="primitive" string:  "string"
                     | @category="primitive" boolean: "bool";

lexical Variable     = @category="variable" Id;
lexical TypeVariable = @category="type" Id;

keyword KW           = "bool"|"number"|"string";
lexical Id           = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]) \ KW;

layout Layout        = [\ \t\n\r]* !>> [\ \t\n\r];
```

Grammar 20: Complex Typed Assignment Restructured

6.10.2 Hypothesis

We expect Grammar 20 to be converted to a TextMate grammar without issues. Accordingly, we expect all inputs to tokenize perfectly according to specification. The reason we expect conversion of Grammar 20 to be better than for Grammar 19 is that the grammar restructure ensures that *Stmt* starts with a regular expression. The non-terminal symbol *TypeWord* can be converted to a single regular expression, while *Type* can not be.

6.10.3 Results

Conversion of Grammar 20 does not generate any errors. Table 21 shows that all input phrases are accurately tokenized. The conversion of this grammar took 22 seconds.

input-name	tokenization grammar	precision
primitive	Program4B	100%
type	Program4B	100%
complexType	Program4B	100%
multiline	Program4B	100%

Table 21: Typed Assignment Restructured Precisions

6.10.4 Discussion

The restructure of the grammar made the conversion process work properly as was expected. This does however show that the conversion pipeline has a lot of room for improvement, since we would like transformations like these to be automated. This experiment shows that the desired tokenization described by

Grammar 19 can be achieved perfectly by a TextMate grammar, but our conversion pipeline is not able to obtain such a grammar on its own.

6.11 Scripting Grammar Experiment

We have seen that conversion can work quite well for rudimentary grammars, and also learned what type structures to look out for. We can not try to construct a simple scripting language grammar and see how well the conversion pipeline deals with this grammar.

6.11.1 Setup

We define a Rascal grammar – here split up into Grammars 21 and 22 – which specifies syntax for a simple scripting language. This grammar is created specifically to be handled well by the conversion pipeline and hence demonstrates a best-case scenario. We convert this grammar to a TextMate grammar and use this on the inputs shown in Table 22. Additionally we will generate a Pygments grammar, such that we can perform syntax highlighting directly from within the LaTeX source code from which this document is generated. Fragment 6 contains the code used for using the Pygments grammar from within LaTeX .

```
syntax Program5 = Stmt*;
syntax Stmt = forIn: For "(" Variable In Exp ")" Stmt
            | forIter: For "(" Exp ";" Exp ";" Exp ")" Stmt
            | iff: If "(" Exp ")" Stmt
            | iffElse: If "(" Exp ")" Stmt Else Stmt
            | "{" Stmt* "}"
            | exp: Exp ";"
            | throww: Throw Exp ";"
            | tryCatch: Try Stmt Catch "(" Variable ")" Stmt
            | tryFinally: Try Stmt Finally Stmt
            | tryCatchFinally: Try Stmt Catch "(" Variable ")" Stmt Finally Stmt
            | ret: Return ";"
            | ret: Return Exp ";";

syntax Exp = var: Variable
            | string: Str
            | booll: Bool
            | nat: Natural
            | call: Exp "(" {Exp ","}* ")"
            | func: Function "(" {Parameter ","}* ")" "{" Stmt* "}"
            | @categoryTerm="keyword.operator" lambda: ("(" {Variable ","}* ")") "=\>" (("{" Stmt*
            ↪  "}") | Exp)
            | @categoryTerm="variable.parameter" index: Exp "[" Exp "]"
            | @categoryTerm="variable.parameter" slice: Exp "[" Exp RangeSep Exp"]"
            | @categoryTerm="variable.parameter" listt: "[" {Exp ","}* "]"
            | brac: "(" Exp ")"
            > @categoryTerm="keyword.operator" not: "!" Exp
            > left (
                  @categoryTerm="keyword.operator" divide: Exp "/" Exp
                | @categoryTerm="keyword.operator" mult: Exp "*" Exp
            )
            > left (
                  @categoryTerm="keyword.operator" subt: Exp "-" Exp
                | @categoryTerm="keyword.operator" add: Exp "+" Exp
            )
            > left (
                  @categoryTerm="keyword.operator" equals: Exp "==" Exp
                | @categoryTerm="keyword.operator" smaller: Exp "\<" Exp
                | @categoryTerm="keyword.operator" greater: Exp "\>" Exp
                | @categoryTerm="keyword.operator" smallerEq: Exp "\<=" Exp
                | @categoryTerm="keyword.operator" greaterEq: Exp "\>=" Exp
            )
            > left (
                  @categoryTerm="keyword.operator" or: Exp "||" Exp
                | @categoryTerm="keyword.operator" and: Exp "&&" Exp
                | @categoryTerm="keyword.operator" inn: Exp "in" Exp
            )
            > left (
                  @categoryTerm="keyword.operator" assign: Exp "=" Exp
                | @categoryTerm="keyword.operator" assignPlus: Exp "+=" Exp
                | @categoryTerm="keyword.operator" assignSubt: Exp "-=" Exp
            );
```

Grammar 21: Scripting Syntax

```
lexical RangeSep = @categoryTerm="keyword.operator" "..";
lexical If = @categoryTerm="keyword" [a-zA-Z0-9] !<< "if";
lexical For = @categoryTerm="keyword" [a-zA-Z0-9] !<< "for";
lexical In = @categoryTerm="keyword.operator" "in";
lexical Else = @categoryTerm="keyword" "else" !>> [a-zA-Z0-9];
lexical Return = @categoryTerm="keyword" [a-zA-Z0-9] !<< "return";
lexical Function = @categoryTerm="entity.name.function" [a-zA-Z0-9] !<< "function";
lexical Throw = @categoryTerm="keyword" [a-zA-Z0-9] !<< "throw";
lexical Try = @categoryTerm="keyword" [a-zA-Z0-9] !<< "try";
lexical Catch = @categoryTerm="keyword" "catch";
lexical Finally = @categoryTerm="keyword" "finally";
lexical Def = @category="variable.parameter" Id;
lexical Variable = @category="variable" Id;
lexical Parameter = @category="variable.parameter" Id;

keyword KW = "for"|"in"|"if"|"true"|"false"|"else"|"return"|"function"|"throw"|"catch"|"finally"|"try";
lexical Id = ([a-zA-Z0-9] !<< [a-zA-Z][a-zA-Z0-9]* !>> [a-zA-Z0-9]) \ KW;
lexical Natural = @category="constant.numeric" [a-zA-Z0-9] !<< [0-9]+ !>> [a-zA-Z0-9];
lexical Bool = @category="constant.other" [a-zA-Z0-9] !<< ("true"|"false") !>> [a-zA-Z0-9];
lexical Str  = @category="string.template" "\"" Char* "\"";
lexical Char = char: ![\\\"$]
             | dollarChar: "$" !>> "{"
             | @categoryTerm="constant.character.escape" escape: "\\"![]
             | @category="meta.embedded.line" @categoryTerm="punctuation.definition.template"
             ↪  embedded: "${" Layout Exp Layout "}";

layout Layout = WhitespaceAndComment* !>> [\ \t\n\r%];
lexical WhitespaceAndComment = [\ \t\n\r]
                             | @category="comment.block" "%" ![%]+ "%"
                             | @category="comment.line" "%%" ![\n]* $;
```

Grammar 22: Scripting Lexical

```
\begin{fragment}
    \caption{Scripting Language Fragment}
    \label{fragment:scripting}
    \begin{minted}{Highlighting/Highlighters/GrammarLexers.py:SimpleRealisticLexer
     ↪  -x}
fib = (i) => {
    if(i<=1) return i;
    else return fib(i-1) + fib(i-2);
};

%% Output the first 10 fibonacci numbers
for(i=0; i<10; i+=1)
    println("fib of ${i} is ${fib(i)}");
    \end{minted}
\end{fragment}
```

Fragment 6: Scripting Language Fragment Source

6.11.2 Hypothesis

We expect Grammar 21 to be converted to a TextMate grammar without issues. Accordingly, we expect all inputs to tokenize perfectly according to specification. The conversion to a Pygments grammar is expected to generate errors, since our grammar does not ensure every scope consists of at most one category. We will

input-name	tokenization grammar	text
assignment	Program5	`something = "hello${true==false}" * 5;`
conditionals	Program5	`if(something==true) %% Some comment something = 45*3; else stuff = (45 + 5) * true;`
loops	Program5	`for(i=0; i<10; i+=2) for(smth in somethingelse) stuff = % inline comment % "${smth} with ${i}";`
arrays	Program5	`something = [true, "false", 12*4]; somethingElse = something[okay()]; thirdItem = something[1..5];`
functions	Program5	`myFunction = function(val1, val2, val3) { return val1; }; something = myFunction(45, myOtherFunction(false && true), "a string ${and(functio)}"); myFunction2 = (something) => 3 * something; myFunction3 = (val1, val2) => { val1 = val1 * 3; return val1 + val2; };`
tryCatch	Program5	`try { something(45); } catch(error) { println(error); } try { something(45); } finally { println("done"); } try { something(45); } catch(error) { println(error); } finally { println("done"); }`

Table 22: Scripting Inputs

simply keep only the outermost categories, which should still provide high-quality syntax highlighting.

6.11.3 Results

Table 23 shows all precisions for the generated TextMate grammar. All inputs were tokenized with full precision. Fragment 7 shows syntax highlighting according to the Pygments syntax highlighter. The conversion to TextMate did not generate any errors, but conversion to the Pygments did result in *disallowedNestedScopes* errors as predicted. The conversion process to both output grammars took 7 minutes and 50 seconds.

input-name	tokenization grammar	precision
assignment	Program5	100%
conditionals	Program5	100%
loops	Program5	100%
arrays	Program5	100%
functions	Program5	100%
tryCatch	Program5	100%

Table 23: Scripting Language Precisions

```
fib = (i) => {
    if(i<=1) return i;
    else return fib(i-1) + fib(i-2);
};

%% Output the first 10 fibonacci numbers
for(i=0; i<10; i+=1)
    println("fib of ${i} is ${fib(i)}");
```

Fragment 7: Scripting Language Fragment

6.11.4 Discussion

This experiment demonstrates that the conversion pipeline works well for sufficiently simple grammars. Targetting multiple formats also behaves as expected. No proper precision tests have been performed for Pygments, Monarch, or Ace grammars, but based on visuals it appears to work as intended.

7 Discussion and Conclusion

In this thesis we tried to solve the practical problem of having to maintain multiple grammars for the same formal language. More specifically, we considered the case of having to maintain one leading context-free grammar, and multiple lexing grammars that are used to provide syntax highlighting in IDEs. The proposed solution to this problem is to augment CFGs with tokenization data, and derive lexing grammars in industry formats from this.

Towards this goal, we defined several instrumental research questions, each of which was discussed in this paper. We started by answering our first question:

> *RQ 1.1: What are the capabilities and limititions of syntax highlighters?*

The lexers TextMate, Monarch, Ace, and Pygments were chosen as targets in order to cover a range of application domains. Together, these lexers cover uses in IDEs, embedded web-editors, and static syntax highlighting for documentation. We found that TextMate grammars are fundamentally different from the other three, having a more declarative nature and allowing scopes – sequences of categories – to be assigned to each character. In contrast, the grammars of the other lexers reveal more information about how the lexer internally works, by means of their *push* and *pop* terminology. These grammars can only assign a single category to each character, but give more freedom in how the internal stack is manipulated. Because of this, the expressivity of these two lexer types is incomparable. We chose to use TextMate as our primary target, which provides the largest restrictions on the structures can be matched. When deriving any of the other three grammar formats, scopes have to be collapsed to a single category, which could eliminate some information if the specification grammar does not ensure every scope already consists of exactly one category. Apart from this, mapping a TextMate grammar to any of the other three grammar formats is straight-forward. After this, we moved to the second question:

> *RQ 1.2: What augmentations can be made to CFGs to allow for specification of syntax highlighting concerns?*

We defined a custom CFG format inspired by Rascal's grammar format. This format is overall simpler than Rascal's format, but provides support for more nuanced scope assignments in productions. We call grammars in this format CGs. In order to support this format, we defined a model for regular expressions that supports positive and negative lookarounds, subtraction, and tag assignments. A corresponding automaton format and conversion scheme was introduced, that allows for reasoning about the language described by a given regular expression. This chapter also covered our next question:

> *RQ 1.3: What form of grammar could easily be mapped to various LG formats?*

This was done by introducing a restricted form of CGs – called a LCG – which allows to intuitively be mapped to TextMate grammars. Finally we answered our last instrumental research question:

> *RQ 1.4: What tokenization preserving transformations can be applied to obtain a LG?*

A conversion pipeline was introduced that operates on CG to obtain a LCG. This pipeline attempts to obtain a LCG that describes a language that is a superset of the original CG. Additionally, it attempts to make the resulting LCG is deterministic. In case this is successful, all tokenizations obtained from the LCG are guaranteed to be identical to those of the specification CG. The pipeline keeps track of errors, and if no errors are generated, it ensures all tokenizations are preserved. If errors are generated, tokenizations may or may not be preserved. This conversion process consists of 4 major parts. The first step attempts to convert grammar constructs into regular expressions, yielding a grammar with an identical language but fewer production components. The second step adds some lookaheads to all regular expressions, increasing the chances of the final grammar to be deterministic. The third step relaxes the grammar in a way that ensures that every production starts with a regular expression. The final and most important step derives a new grammar from the original grammar with the desired LCG shape. This is done by defining new non-terminal symbols based on non-terminals from the original grammar, to which rewrite rules are applied that ensure the correct structure is reached. After a LCG is obtained, a mapping step is performed to any of the four target grammar formats. For this mapping, two intermediate grammar formats are used to represent

the different lexing grammar architectures. This process also has to correct for some small discrepancies between the final LCG and the exact LG formats. Finally, we attempted to answer our primary research question:

> *RQ 1: Can a static grammar transformation pipeline effectively be used to derive a CFG and a LG from an augmented CFG?*

This was done based on some experiments on our conversion pipeline. It however came in the form of hand-crafted test-cases, used to illustrate the capabilities and limitations of this approach. This revealed one major limitation of the overlap combining technique, which prevents this pipeline in its current state to be used for many existing grammars.

7.1 Contributions

This thesis explored a practical topic for which little formal research exists so far. Because of this, we touched on various topics that all play a role in reaching the goal of deriving lexing grammars. We provided the following contributions:

- Syntax, semantics, and tools surrounding regular languages with tags and contexts in Section 3.1:
  - The notion of expressing languages with an alphabet Σ, with tag universe T, and contexts as a subset of the triple $(\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^* \times (\Sigma \times \mathcal{P}(T))^*$.
  - Syntax, semantics, and transformations (in Section 4) for TLREs.
  - Definitions, semantics, and operations for TCNFAs.
  - A mapping from TLREs to TCNFAs expressing identical languages.
  - Rascal implementations of all these concepts.
- Syntax and semantics of various grammar models in Sections 3.2 and 4.5:
  - Syntax and semantics of CGs, the subset of LCGs and choice-free LCGs.
  - Syntax and semantics of Scope grammars, modeling TextMate grammars.
  - Syntax and semantics of PDA grammars, modeling Monarch, Ace, and Pygments grammars.
- A conversion pipeline from CFGs to TextMate, Monarch, Ace, and Pygments grammars in Chapters 3.2, 5, and 6:
  - Various rewrite rules on CGs to obtain LCGs.
  - A check on LCG to determine whether it is free of choices.
  - Mapping from LCG to Scope grammars and PDA grammars, and in turn TextMate, Monarch, Ace, and Pygments grammars.
  - Conjectures relating to the completeness and soundness of the conversion pipeline.
  - A Rascal implementation of this complete pipeline.
  - A testing and experimentation procedure to verify this pipeline.
- A Rascal value exploration tool described in Section 5.1.

7.2 Limitations

We have shown that the conversion approach has potential, but were not able to adequately answer our primary research question: can a static grammar transformation pipeline effectively be used to derive a CFG and a LG from an augmented CFG? To answer this question negatively, a strong argument would have to be found as to why this would not be possible. We instead provided some evidence towards this being possible, but were not able to refine the approach enough to make it adequately handle CFG of existing languages. We identified three major limitations with the current approach:

- Prefix conversion removes structural data
- Consecutive non-terminal combining removes structural data
- Overlap combining can lead to exponential blow-up

The first two of these limitations are both forms of grammar relaxation, which might lead to non-determinism in the final grammar. There are cases where this leads to non-determinism even though an adequate deterministic LCG does exist. These problems are relatively minor however, since they do not prevent creation of LGs, only reduce their quality. The last limitation is however a major one. Such exponential blow-up can cause the entire conversion to not finish within a reasonable amount of time, making the algorithm useless in these cases.

Besides limitations with the conversion pipeline itself, there are also limitations regarding the applicability of our regular expression formalism in other domains. As we described before, our notion of tags does not fully reflect the behavior of capture groups within existing regular expression engines. Use of this formalism to analyze regular expressions within existing software is limited to a subset of all possible regular expressions. Only for regular expressions without optional captures or repeated captures, our model properly reflects the semantics of regular expression engines. We do however believe that many real-world expressions do belong to this class.

Even though the results from this research were not perfect, the limitations we discovered are not necessarily fundamental shortcomings. With more research, these problems might get solved. We have several ideas for how to achieve this, which we will discuss in the next section.

7.3 Future Directions

There are many directions that this research can be expanded on. Most trivially, the open proof obligations could be addressed, and the discussed limitations could be resolved.

One possible way of improving prefix conversion, is to perform merging of productions instead. When encountering a production $A \to B\ \alpha$, every productions of B could be merged with the suffix α when copying it. This keeps more structural information in the grammar, and might prevent non-determinism created by our current more relaxed transformation. Similar schemes could also be explored for making consecutive non-terminal combining retain more structural information. One danger of retaining more information however, is that it could result in larger grammars, including possibility of exponential blow-up. It is worth researching this in more depth.

Overlap combining causes several distinct problems. The problem related to exponential blow-up due to optional suffixes could be solved by relaxing these sequences at an earlier stage assuming this relaxation does not cause non-determinism. The difficulty lies in predicting whether any such relaxation would cause determinism later in the conversion process or not. Heuristics based on existing languages could be used for this, or more clever analytical techniques could be explored. Additionally, more clever overlap combining techniques could be explored all-together, since this step is amongst the most important steps for deriving an accurate tokenizer.

Apart from solving the limitations of the described approach, there is also a lot of room for expanding on it. After the limitations have been decreased, real world performance of the approach should be analyzed. This involves augmenting existing CFGs with tokenization data, and trying to use the conversion pipeline to obtain corresponding LGs. Assuming this conversion is not fully accurate, one could investigate the real-world accuracy of a grammar using a large sample set of real source-code files. Besides measuring accuracy, it would also be worth developing the other measures we vaguely touched upon. Especially robustness is relevant for syntax-highlighters used in IDEs, since code is often in an invalid state while typing.

One could also explore a different lexing grammar target. We decided to accept the limitations imposed by TextMate, but in cases where TextMate grammars are of no importance, better LCG could be derived.

As mentioned before, we believe that our regular expression model could promise useful outside this research. The mismatch between our tag system and capture groups of regular expression engines does however limit applicability. Therefore it might be worth investigating an alternative way of encoding capture groups in languages and NFAs. In earlier stages of this research, we already briefly explored the idea

of using boundary tokens in the language to specify capture groups. This is similar to how boundary tokens are used to separate the prefix, main match, and suffix. We dropped this idea in favor of tags, due to tags being a better fit for our specific purpose, and seemingly easier to work with. The main difficulty we encountered with using boundary tokens, is that their ordering is difficult to deal with in the NFA theory. For instance if two capture groups are directly nested, always capturing the same matches, the ordering of their boundary tokens in the described language becomes irrelevant. A regular expression model should adequately capture this. This is something that we had trouble with, but a simple theory around the idea of boundary tokens might exist nevertheless, and is worth exploring further.

Besides improving our regular expression model, we believe the current version can already be useful for several applications. As discussed earlier, it might be of use for automated analysis and transformations, but it might also prove useful for creating linear time regex engine matchers that support lookarounds. The Rust language uses a regular expression engine that gives linear time searching guarantees, but sacrifices the lookaround feature to achieve this. Our regular expression model and its corresponding automata, might prove useful for facilitating efficient searches that do support lookarounds. This is something that has not been explored in our research at all, and thus might not be feasible, but the existence of TCDFAs hints at possible linear time use-cases.

Overall, many of the explored topics could be expanded upon and even possibly find uses in other applications.

7.4 Conclusion

In conclusion, this thesis explored the idea of deriving lexing grammars from context-free grammars augmented with tokenization data. This was done by introducing a formal conversion grammar model to operate on, together with a formal regular expression model that supports lookarounds and tags. We introduced a grammar transformation pipeline that operates on conversion grammars, resulting in a final conversion grammar form that closely resembles the structure of TextMate grammars. This grammar is then mapped to the target lexing grammars. This conversion pipeline is not able to always guarantee that the tokenization described by the specification conversion grammar is identical to the one described by the output grammar for all inputs. However, based on Conjectures 2 and 3 it does guarantee that if all conditions are met (and thus no errors are generated), all tokenizations are fully retained.

We described the concepts used to achieve this, and implemented them in Rascal. This implementation allowed us to perform several experiments, from which various limitations were found. We have also shown that despite these limitations, the transformation pipeline can convert some simple – yet realistic – grammars with full precision. Finally we discussed these shortcomings, and provided several suggestions that might remedy these. We believe that additional research will yield a pipeline that achieves our goal and fully automatically generates lexing grammars for any new language one may develop.

References

[1] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[2] Jurgen J. Vinju. Comparing Bottom-Up with Top-Down Parsing Architectures for the Syntax Definition Formalism from a Disambiguation Standpoint. In *Eelco Visser Commemorative Symposium (EVCS 2023)*, volume 109, 2023.

[3] Allan Odgaard. Textmate grammar documentation. 2018. `https://macromates.com/manual/en/language_grammars`.

[4] K.Kosako. Oniguruma regular expressions. 2015. `https://github.com/kkos/oniguruma`.

[5] Daan Leijen. Monarch grammar documentation. 2022. `https://microsoft.github.io/monaco-editor/monarch.html`.

[6] Ace grammar documentation. 2012. `https://ace.c9.io/#nav=higlighter`.

[7] Georg Brandl. Pygments grammar documentation. 2015. `https://pygments.org/docs/lexerdevelopment`.

[8] SC Kleene. REPRESENTATION OF EVENTS IN NERVE NETS AND FINITE AUTOMATA^1. *Automata Studies: Annals of Mathematics Studies. Number 34*, (34):3, 1956. Publisher: Princeton University Press.

[9] Takayuki Miyazaki and Yasuhiko Minamide. Derivatives of Regular Expressions with Lookahead. *Journal of Information Processing*, 27(0):422–430, 2019.

[10] Martin Berglund, Brink Van Der Merwe, and Steyn Van Litsenborgh. Regular Expressions with Lookahead. *JUCS - Journal of Universal Computer Science*, 27(4):324–340, April 2021.

[11] V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 181–187, A Curuna, Spain, 2000. IEEE Comput. Soc.

[12] From Regular Expressions to Automata. In Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors, *Compilers: principles, techniques, & tools*, pages 152–166. Pearson Addison-Wesley, Boston Munich, 2. ed., pearson internat. ed edition, 2007.

[13] Alberto Pettorossi. Finite Automata and Regular Grammars. In Alberto Pettorossi, editor, *Automata Theory and Formal Languages: Fundamental Notions, Theorems, and Techniques*, Undergraduate Topics in Computer Science, pages 25–100. Springer International Publishing, Cham, 2022.

[14] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Principle of Programming Languages (POPL)*, pages 457–468, Roma, Italy, January 2013. ACM.

[15] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[16] Ganesh Gopalakrishnan. Context-free Languages. In *Computation Engineering: Applied Automata Theory and Logic*, pages 217–244. Springer US, Boston, MA, 2006.

[17] Jacques Sakarovitch. THE SIMPLEST POSSIBLE MACHINE. In ReubenTranslator Thomas, editor, *Elements of Automata Theory*, pages 49–216. Cambridge University Press, 2009.

[18] Norman Ramsey. Unparsing expressions with prefix and postfix operators. *Software: Practice and Experience*, 28(12):1327–1356, 1998.

[19] FIRST and FOLLOW. In Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors, *Compilers: principles, techniques, & tools*, pages 220–222. Pearson Addison-Wesley, Boston Munich, 2. ed., pearson internat. ed edition, 2007.

Glossary

Ace is a embeddable code editor written in JavaScript. It serves as the primary editor for the Cloud9 IDE. 4, 5, 10, 11, 48, 57, 60, 62, 91, 92, 93

alphanumeric is a term used to describe a combination of alphabetic and numeric characters. 2

Github is a cloud-based version control service. 60, 63

IntelliJ IDEA is a cross-platform IDE, focused on java development. 1

Java is a general-purpose programming language with an object oriented paradigm focusing on cross-platform usage. 82

JavaScript is the default programming language used in the front-end of websites. 6, 8, 10, 11, 57, 97, 98

Minted is a LaTeX package that uses Pygments for syntax highlighting of code fragments in LaTeX. 5

Monaco is a texteditor used by vscode. 5

Monarch is a lexer used by monaco for syntax highlighting. 4, 5, 8, 10, 11, 48, 57, 60, 62, 91, 92, 93

Node.js is a JavaScript runtime that allows JavaScript to be used outside an web browser. 62

Pygments is a syntax highlighter written in python, that supports various output formats including LaTeX. 4, 5, 11, 48, 56, 57, 60, 62, 87, 89, 90, 91, 92, 93, 97

Python is a programming language used by many engineers, with a focus on data processing. 11, 56

Rascal is a meta-programming language. 2, 3, 4, 13, 28, 29, 52, 58, 60, 62, 63, 66, 69, 73, 76, 78, 79, 80, 82, 83, 85, 87, 92, 93, 95, 105, 111

Rust is a general-purpose programming language that emphasizes performance and type safety. 95

Sublime Text is a cross-platform code editor. 1, 5

TextMate is a text editor for macOS with various language features. 1, 4, 5, 6, 8, 9, 11, 12, 28, 48, 53, 60, 62, 66, 68, 69, 73, 75, 76, 78, 79, 80, 83, 85, 86, 87, 89, 90, 92, 93, 94, 95

tree-sitter is a parser generator tool, which has first-party syntax highlighting support. 1, 3

TypeScript is a superset of JavaScript that adds a static type checking system and compiles down to JavaScript. 62

Acronyms

API Application Programming Interface. 3

CFG Context-free Grammar. 1, 2, 3, 4, 6, 8, 13, 25, 26, 27, 33, 46, 58, 65, 66, 92, 93, 94

CG Conversion Grammar. 3, 4, 13, 25, 26, 27, 28, 29, 31, 39, 48, 58, 62, 63, 92, 93

CSS Cascading Style Sheets. 8, 10

DFA Deterministic Finite Automaton. 17, 18, 19, 111, 112, 113

DPDA Deterministic Pushdown Automaton. 9, 10, 11, 12

IDE Integrated Development Environment. 1, 3, 5, 92, 94, 97

JSON JavaScript Object Notation. 6, 7, 9, 56, 57, 68

LCG Lexing Conversion Grammar. 4, 29, 31, 32, 38, 39, 46, 47, 48, 50, 51, 53, 55, 56, 58, 60, 69, 75, 92, 93, 94

LG Lexing Grammar. 1, 2, 3, 4, 7, 13, 29, 31, 32, 36, 48, 49, 59, 65, 66, 92, 93, 94

NFA Non-deterministic Finite Automaton. 3, 17, 18, 19, 47, 48, 94, 95

PDA Pushdown Automaton. 4, 31, 48, 53, 54, 55, 56, 57, 93

TCDFA Tagged Contextualized Deterministic Finite Automaton. 18, 19, 24, 48, 95, 107, 110, 111, 112, 113

TCNFA Tagged Contextualized Non-deterministic Finite Automaton. 3, 18, 19, 20, 21, 24, 35, 36, 40, 41, 42, 43, 48, 52, 58, 60, 63, 93, 99, 100, 105, 106, 107, 110, 111, 112, 113

TLRE Tagged Lookaround Regular Expressions. 3, 93

VSCode Visual Studio Code. 1, 3, 5, 62

Appendix

A Complete TCNFA conversion

This appendix contains all required definitions for inductively obtaining a TCNFA from a given regular expression. First we will briefly discuss the concept behind iteration TCNFAs representing the Kleene closure of a TCNFA, as well as negative lookarounds, and finally we will list all definitions.

For the remainder of this chapter, let r_1 and r_2 be two regular expressions, with their corresponding TCNFAs:

- $(Q_1^p, Q_1^w, Q_1^s, \Sigma, T, R_1, i_1, F_1) = TCNFA(r_1)$
- $(Q_2^p, Q_2^w, Q_2^s, \Sigma, T, R_2, i_2, F_2) = TCNFA(r_2)$

Concatenation and lookaround TCNFAs are constructed by creating a product automaton of both arguments, and using the match-start and match-end transitions to synchronize them. We could consider $TCNFA(r_1^+)$ to be the infinite alternation of all possible length concatenations: $TCNFA(r_1^+) = TCNFA(r_1 + r_1 r_1 + r_1 r_1 r_1 + \cdots)$. When considering a single option from these alternations, the corresponding TCNFA is constructed as the concatenation of n copies of r_1 for some constant n. We can call the corresponding TCNFA a n-concatenation automaton. This means that every state in a n-concatenation automaton is essentially a n-tuple consisting of n states of $TCNFA(r_1)$. In these states, the exact ordering of elements is irrelevant, only what sub-states are contained in them matters. Therefore the state of a n-concatenation automaton could be represented by multi-set containing exactly n sub-states. The initial state for each of these automata would be a multi-set of n copies of i_1 .Now we would like $TCNFA(r_1^+)$ to simulate an infinite number of these n-concatenation automaton, one for every possible $n \in \mathbb{N}^+$. This can be done by creating states that abstract over the exact number of sub-states contained in them. For this we use a set of sub-states. Such a set state represents all multi-set states of n-concatenation automata that are constructed from only sub-states in this set. Consider the state $\{i_1\}$, this would represent all multi-set states $\{i_1\}, \{i_1, i_1\}, \{i_1, i_1, i_1\}, \cdots$. When simulating transitions, we have to take special care to consider that any sub-state present in the set represents this state being contained one or more times. The only exception to this is any sub-state $m \in Q_1^w$, since any concatenation can only capture one character of every sub-automata at a time. When taking a character transition from such a set-state, every sub-state in this state has to to make such a character-transition. If a sub-state can make multiple transitions for the same character, it can make multiple of these at the same time, since it is unknown how many copies of this sub-state we are in. We will now formalize this idea with a proper definition.

Let $\forall\exists\ c_1;\ c_2\ .\ q$ be shorthand for $(\forall\ c_1\ .\ \exists\ c_2\ .\ q) \wedge (\forall\ c_2\ .\ \exists\ c_1\ .\ q)$. Then we can define $TCNFA(r_1^+) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= \mathcal{P}(Q_1^p) \\
Q_w &= \{P \cup \{m\} \cup S \mid m \in Q_1^w \wedge P \subseteq Q_1^p \wedge S \subseteq Q_1^s\} \\
Q_s &= \mathcal{P}(Q_1^s) \\
R &= \{S \xrightarrow{\epsilon} S' \mid S, S' \in Q_p \cup Q_w \cup Q_s \wedge S \neq S' \wedge \forall\exists\ s \in S;\ s' \in S'\ .\ s = s' \vee s \xrightarrow{\epsilon} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{(a,T')} S' \mid S, S' \in Q_p \cup Q_w \cup Q_s \wedge (\forall\exists\ s \in S;\ s' \in S'\ .\ s \xrightarrow{(a,T_s)} s' \in R_1 \wedge T_s \subseteq T') \\
&\qquad\qquad \wedge \forall\ t \in T'\ .\ \exists\ s \in S, s' \in S', s \xrightarrow{(a,T_s)} s' \in R_1\ .\ t \in T_s\} \\
&\quad \cup \{S \xrightarrow{\langle} S' \mid S \in Q_p \wedge S' \in Q_w \wedge \forall\exists\ s \in S;\ s' \in S'\ .\ s = s' \vee s \xrightarrow{\langle} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{\rangle} S' \mid S \in Q_w \wedge S' \in Q_s \wedge \forall\exists\ s \in S;\ s' \in S'\ .\ s = s' \vee s \xrightarrow{\rangle} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{\epsilon} S' \mid \exists\ p \in Q_1^p, P \subseteq Q_1^p, s \in Q_1^s, S \subseteq Q_1^s,\ m, m' \in Q_1^w \\
&\qquad\qquad .\ S = (\{p, m\} \cup P \cup S) \wedge S' = (\{m', s\} \cup P \cup S) \wedge p \xrightarrow{\langle} m' \in R_1 \wedge m \xrightarrow{\rangle} s \in R_1\} \\
i &= \{i_1\} \\
F &= \mathcal{P}(F_1)
\end{aligned}
$$

Defining a negative lookahead TCNFA requires us to obtain the complement of $TCNFA(r_2)$. Before obttaining the complement of $TCNFA(r_2)$ we replace all transition tag set by $\emptyset$, and we use $\emptyset$ as the tags-universe. Moreover, the match-end transition has to be replaced by an ϵ transition before obtaining this complement. After this is done, a negative lookahead is constructed almost identically to a positive lookahead, simply using the negated lookahead automaton. If the match-end transition were not remove before computing the complement of $TCNFA(r_2)$ but instead was removed when computing the product automaton, the resulting automaton would describe words for which there exists is a suffix that is not in $\mathcal{L}(r_2)$, rather than words for which there does not exists any suffix that is in $\mathcal{L}(r_2)$. Let $(\overline{Q_2^p}, \overline{Q_2^w}, \overline{Q_2^s}, \Sigma, \emptyset, \overline{R_2}, \overline{i_2}, \overline{F_2})$ be the complement TCNFA of $TCNFA(r_2)$ for which the match-end transition was replaced by an epsilon transition before computing the complement. Now we can define the negative lookahead as $TCNFA(r_1 \not> r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= Q_1^p \times \overline{Q_2^p} \\
Q_w &= Q_1^w \times \overline{Q_2^p} \\
Q_s &= Q_1^s \times (\overline{Q_2^w} \cup \overline{Q_2^s}) \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (\overline{Q_2^p} \cup \overline{Q_2^w} \cup \overline{Q_2^s}) \wedge s_1 \xrightarrow{\epsilon} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (\overline{Q_2^p} \cup \overline{Q_2^w} \cup \overline{Q_2^s}) \wedge s_2 \xrightarrow{\epsilon} s_2' \in \overline{R_2}\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (\overline{Q_2^p} \cup \overline{Q_2^w} \cup \overline{Q_2^s}) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in \overline{R_2}\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^p \wedge s_2 \in \overline{Q_2^p} \wedge s_1 \xrightarrow{\langle} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in \overline{Q_2^p} \wedge s_1 \xrightarrow{\rangle} s_1' \in R_1 \wedge s_2 \xrightarrow{\langle} s_2' \in \overline{R_2}\} \\
i &= (i_1, i_2) \\
F &= F_1 \times F_2
\end{aligned}
$$

For later usage, we will define $(\overline{Q_1^p}, \overline{Q_1^w}, \overline{Q_1^s}, \Sigma, \emptyset, \overline{R_1}, \overline{i_1}, \overline{F_1})$ be the complement TCNFA of $TCNFA(r_1)$ for which the match-start transition was replaced by an epsilon transition before computing the complement.

Below is an overview of all TCNFA constructors:

$$
\begin{aligned}
&TCNFA(\mathbf{0}) = (\{p\}, \emptyset, \emptyset, \Sigma, T, \emptyset, p, \emptyset) \\
&TCNFA(\mathbf{1}) = (\{p\}, \{m\}, \{s\}, \Sigma, T, \{p \xrightarrow{\langle} m, m \xrightarrow{\rangle} s\} \cup \{p \xrightarrow{(a, \emptyset)} p, m \xrightarrow{(a, \emptyset)} m, s \xrightarrow{(a, \emptyset)} s \mid a \in \Sigma\}, p, \{s\}) \\
&TCNFA(\epsilon) = (\{p\}, \{m\}, \{s\}, \Sigma, T, \{p \xrightarrow{\langle} m, m \xrightarrow{\rangle} s\} \cup \{p \xrightarrow{(a, \emptyset)} p, s \xrightarrow{(a, \emptyset)} s \mid a \in \Sigma\}, p, \{s\}) \\
&TCNFA(a) = (\{p\}, \{m_1, m_2\}, \{s\}, \Sigma, T, \{p \xrightarrow{\langle} m_1, m_1 \xrightarrow{a} m_2, m_2 \xrightarrow{\rangle} s\} \cup \{p \xrightarrow{(a, \emptyset)} p, s \xrightarrow{(a, \emptyset)} s \mid a \in \Sigma\}, p, \{s\})
\end{aligned}
$$

$TCNFA(r_1{+}r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= \{p\} \cup \{(p_1, \epsilon) \mid p_1 \in Q_1^p\} \cup \{(\epsilon, p_2) \mid p_2 \in Q_2^p\} \\
Q_w &= \{(m_1, \epsilon) \mid m_1 \in Q_1^w\} \cup \{(\epsilon, m_2) \mid m_2 \in Q_2^w\} \\
Q_s &= \{s\} \cup \{(s_1, \epsilon) \mid s_1 \in Q_1^s\} \cup \{(\epsilon, s_2) \mid s_2 \in Q_2^s\} \\
R &= \{p \xrightarrow{\epsilon} (p_1, \epsilon) \mid p_1 \in Q_1^p\} \\
&\quad \cup \{p \xrightarrow{\epsilon} (\epsilon, p_2) \mid p_2 \in Q_2^p\} \\
&\quad \cup \{(q_1, \epsilon) \xrightarrow{c} (q_1', \epsilon) \mid q_1 \xrightarrow{c} q_1' \in R_1\} \\
&\quad \cup \{(\epsilon, q_2) \xrightarrow{c} (\epsilon, q_2') \mid q_2 \xrightarrow{c} q_2' \in R_2\} \\
&\quad \cup \{(s_1, \epsilon) \xrightarrow{\epsilon} s \mid s_1 \in Q_1^s\} \\
&\quad \cup \{(\epsilon, s_2) \xrightarrow{\epsilon} s \mid s_2 \in Q_2^s\} \\
i &= p \\
F &= \{s\}
\end{aligned}
$$

$TCNFA(r_1 r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= Q_1^p \times Q_2^p \\
Q_w &= (Q_1^w \times Q_2^p) \cup (Q_1^s \times Q_2^w) \\
Q_s &= Q_1^s \times Q_2^s \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{\epsilon} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_2 \xrightarrow{\epsilon} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^p \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\langle} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\rangle} s_1' \in R_1 \wedge s_2 \xrightarrow{\langle} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1, s_2') \mid s_1 \in Q_1^s \wedge s_2 \in Q_2^w \wedge s_2 \xrightarrow{\rangle} s_2' \in R_2\} \\
i &= (i_1, i_2) \\
F &= F_1 \times F_2
\end{aligned}
$$

$TCNFA(r_1{>}r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= Q_1^p \times Q_2^p \\
Q_w &= Q_1^w \times Q_2^p \\
Q_s &= Q_1^s \times (Q_2^w \cup Q_2^s) \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{\epsilon} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_2 \xrightarrow{\epsilon} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^p \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\langle} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\rangle} s_1' \in R_1 \wedge s_2 \xrightarrow{\langle} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in Q_1^s \wedge s_2 \in Q_2^w \wedge s_2 \xrightarrow{\rangle} s_2' \in R_2\} \\
i &= (i_1, i_2) \\
F &= F_1 \times F_2
\end{aligned}
$$

$TCNFA(r_1{<}r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= (Q_1^p \cup Q_1^w) \times Q_2^p \\
Q_w &= Q_1^s \times Q_2^w \\
Q_s &= Q_1^s \times Q_2^s \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{\epsilon} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_2 \xrightarrow{\epsilon} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in Q_1^p \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\langle} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\rangle} s_1' \in R_1 \wedge s_2 \xrightarrow{\langle} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1, s_2') \mid s_1 \in Q_1^s \wedge s_2 \in Q_2^w \wedge s_2 \xrightarrow{\rangle} s_2' \in R_2\} \\
i &= (i_1, i_2) \\
F &= F_1 \times F_2
\end{aligned}
$$

$TCNFA(r_1 \not> r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= Q_1^p \times \overline{Q_2^p} \\
Q_w &= Q_1^w \times \overline{Q_2^p} \\
Q_s &= Q_1^s \times (\overline{Q_2^w} \cup \overline{Q_2^s}) \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (\overline{Q_2^p} \cup \overline{Q_2^w} \cup \overline{Q_2^s}) \wedge s_1 \xrightarrow{\epsilon} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (\overline{Q_2^p} \cup \overline{Q_2^w} \cup \overline{Q_2^s}) \wedge s_2 \xrightarrow{\epsilon} s_2' \in \overline{R_2}\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (\overline{Q_2^p} \cup \overline{Q_2^w} \cup \overline{Q_2^s}) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in \overline{R_2}\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^p \wedge s_2 \in \overline{Q_2^p} \wedge s_1 \xrightarrow{\langle} s_1' \in R_1\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in \overline{Q_2^p} \wedge s_1 \xrightarrow{\rangle} s_1' \in R_1 \wedge s_2 \xrightarrow{\langle} s_2' \in \overline{R_2}\} \\
i &= (i_1, \overline{i_2}) \\
F &= F_1 \times \overline{F_2}
\end{aligned}
$$

$TCNFA(r_1 \not< r_2) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= (\overline{Q_1^p} \cup \overline{Q_1^w}) \times Q_2^p \\
Q_w &= \overline{Q_1^s} \times Q_2^w \\
Q_s &= \overline{Q_1^s} \times Q_2^s \\
R &= \{(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2) \mid s_1 \in (\overline{Q_1^p} \cup \overline{Q_1^w} \cup \overline{Q_1^s}) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{\epsilon} s_1' \in \overline{R_1}\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\epsilon} (s_1, s_2') \mid s_1 \in (\overline{Q_1^p} \cup \overline{Q_1^w} \cup \overline{Q_1^s}) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_2 \xrightarrow{\epsilon} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{(a, T_1 \cup T_2)} (s_1', s_2') \\
&\qquad \mid s_1 \in (\overline{Q_1^p} \cup \overline{Q_1^w} \cup \overline{Q_1^s}) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{(a, T_1)} s_1' \in \overline{R_1} \wedge s_2 \xrightarrow{(a, T_2)} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\langle} (s_1', s_2) \mid s_1 \in Q_1^w \wedge s_2 \in Q_2^p \wedge s_1 \xrightarrow{\rangle} s_1' \in \overline{R_1} \wedge s_2 \xrightarrow{\langle} s_2' \in R_2\} \\
&\quad \cup \{(s_1, s_2) \xrightarrow{\rangle} (s_1, s_2') \mid s_1 \in Q_1^s \wedge s_2 \in Q_2^w \wedge s_2 \xrightarrow{\rangle} s_2' \in R_2\} \\
i &= (\overline{i_1}, i_2) \\
F &= \overline{F_1} \times F_2
\end{aligned}
$$

$TCNFA(r_1^+) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= \mathcal{P}(Q_1^p) \\
Q_w &= \{P \cup \{m\} \cup S \mid m \in Q_1^w \wedge P \subseteq Q_1^p \wedge S \subseteq Q_1^s\} \\
Q_s &= \mathcal{P}(Q_1^s) \\
R &= \{S \xrightarrow{\epsilon} S' \mid S, S' \in Q_p \cup Q_w \cup Q_s \wedge S \neq S' \wedge \forall\exists\ s \in S;\ s' \in S' \,.\, s = s' \vee s \xrightarrow{\epsilon} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{a} S' \mid S, S' \in Q_p \cup Q_w \cup Q_s \wedge \forall\exists\ s \in S;\ s' \in S' \,.\, s \xrightarrow{a} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{\langle} S' \mid S \in Q_p \wedge S' \in Q_w \wedge \forall\exists\ s \in S;\ s' \in S' \,.\, s = s' \vee s \xrightarrow{\langle} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{\rangle} S' \mid S \in Q_w \wedge S' \in Q_s \wedge \forall\exists\ s \in S;\ s' \in S' \,.\, s = s' \vee s \xrightarrow{\rangle} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{\epsilon} S' \mid \exists\ p \in Q_1^p, P \subseteq Q_1^p, s \in Q_1^s, S \subseteq Q_1^s,\ m, m' \in Q_1^w \\
&\qquad\qquad . S = (\{p, m\} \cup P \cup S) \wedge S' = (\{m', s\} \cup P \cup S) \wedge p \xrightarrow{\langle} m' \in R_1 \wedge m \xrightarrow{\rangle} s \in R_1\} \\
i &= \{i_1\} \\
F &= \mathcal{P}(F_1)
\end{aligned}
$$

$TCNFA(r_1^+) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= \mathcal{P}(Q_1^p) \\
Q_w &= \{P \cup \{m\} \cup S \mid m \in Q_1^w \wedge P \subseteq Q_1^p \wedge S \subseteq Q_1^s\} \\
Q_s &= \mathcal{P}(Q_1^s) \\
R &= \{S \xrightarrow{\epsilon} S' \mid S, S' \in Q_p \cup Q_w \cup Q_s \wedge S \neq S' \wedge \forall\exists\ s \in S;\ s' \in S' \,.\, s = s' \vee s \xrightarrow{\epsilon} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{(a,T')} S' \mid S, S' \in Q_p \cup Q_w \cup Q_s \wedge (\forall\exists\ s \in S;\ s' \in S' \,.\, s \xrightarrow{(a,T_s)} s' \in R_1 \wedge T_s \subseteq T') \\
&\qquad\qquad \wedge \forall\ t \in T' \,.\, \exists\ s \in S, s' \in S', s \xrightarrow{(a,T_s)} s' \in R_1 \,.\, t \in T_s\} \\
&\quad \cup \{S \xrightarrow{\langle} S' \mid S \in Q_p \wedge S' \in Q_w \wedge \forall\exists\ s \in S;\ s' \in S' \,.\, s = s' \vee s \xrightarrow{\langle} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{\rangle} S' \mid S \in Q_w \wedge S' \in Q_s \wedge \forall\exists\ s \in S;\ s' \in S' \,.\, s = s' \vee s \xrightarrow{\rangle} s' \in R_1\} \\
&\quad \cup \{S \xrightarrow{\epsilon} S' \mid \exists\ p \in Q_1^p, P \subseteq Q_1^p, s \in Q_1^s, S \subseteq Q_1^s,\ m, m' \in Q_1^w \\
&\qquad\qquad . S = (\{p, m\} \cup P \cup S) \wedge S' = (\{m', s\} \cup P \cup S) \wedge p \xrightarrow{\langle} m' \in R_1 \wedge m \xrightarrow{\rangle} s \in R_1\} \\
i &= \{i_1\} \\
F &= \mathcal{P}(F_1)
\end{aligned}
$$

$TCNFA((\langle t \rangle r_1)) = (Q_p, Q_w, Q_s, \Sigma, T, R, i, F)$ where:

$$
\begin{aligned}
Q_p &= Q_1^p \\
Q_w &= Q_1^w \\
Q_s &= Q_1^s \\
R &= \{s \xrightarrow{\epsilon} s' \mid s \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s \xrightarrow{\epsilon} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{\langle} s' \mid s \in Q_1^p \wedge s \xrightarrow{\langle} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{\rangle} s' \mid s \in Q_1^w \wedge s \xrightarrow{\rangle} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{(a,T_1)} s' \mid s \in (Q_1^p \cup Q_1^s) \wedge s \xrightarrow{(a,T_1)} s' \in R_1\} \\
&\quad \cup \{s \xrightarrow{(a,\{t\}\cup T_1)} s' \mid s \in Q_1^w \wedge s \xrightarrow{(a,T_1)} s' \in R_1\} \\
i &= i_1 \\
F &= F_1
\end{aligned}
$$

B Regular Expressions Implementation

In our grammars, characters are represented using unicode. Character classes are a core part of Rascal and therefore are also supported by our TCNFAs representation. Certain TCNFA may have character transitions that accepts hundreds of thousands of characters, each of which would require an independent transition if no character classes were used. Therefore we use the same character classes as both the regular expressions and Rascal's grammar already use. These character classes are ordered sets of character ranges, such that any character that falls in one of these ranges cis accepted by the transition. Similarly we make use of tags classes to combine multiple options of tag sets. These classes are a bit simpler, and are merely sets of tags sets. Then a transitions in a TCNFA is applicable for any pair of character and tag set if the character is within one of the given character ranges and the tag set is included in the tags class. This, together with the epsilon and matchStart/matchEnd transitions, can be represented using the Rascal code in Fragment 8.

```
alias Char = int;

alias CharClass = list[CharRange];
data CharRange = range(Char begin, Char end); // where both begin and end are
 ↪  inclusive

alias TagSet = set[value];
alias TagClass = set[TagSet];

data TransitionCondition = characterCon(CharClass, TagClass)
                         | matchStartCon()
                         | matchEndCon()
                         | epsilon();
data PhraseSymbol = character(Char, TagSet)
                  | matchStart()
                  | matchEnd();

bool applies(TransitionCondition condition, PhraseSymbol symbol) {
    switch(symbol) {
        case matchStart():       return matchStartCon() := condition;
        case matchEnd():         return matchEndCon() := condition;
        case character(cc, tc):  return characterCon(c, t) := condition
                                        && includes(cc, c)
                                        && includes(tc, t);
        default: return false;
    }
}
bool includes(CharClass cc, Char c) = any(range(begin, end) <- cc, begin <= c && c
 ↪  <= end);
bool includes(TagClass tc, TagSet t) = t in tc;
```

Fragment 8: TCNFA Transition Implementation

This approach significantly decreases the sizes of TCNFAs that accept many characters, but it also introduces quite some extra complexity that should be dealt with in all of the algorithms dealing with TCNFAs. Lets first consider the product automaton approach that is at the the heart of converting concatenation, lookaround, and reservation regular expressions to TCNFAs. The character transitions of these automata only had to consider 1 character at a time before, and were defined as follows:

$$\{(s_1, s_2) \xrightarrow{(a,T_1 \cup T_2)} (s_1', s_2') \mid s_1 \in (Q_1^p \cup Q_1^w \cup Q_1^s) \wedge s_2 \in (Q_2^p \cup Q_2^w \cup Q_2^s) \wedge s_1 \xrightarrow{(a,T_1)} s_1' \in R_1 \wedge s_2 \xrightarrow{(a,T_2)} s_2' \in R_2\}$$

When using classes, we have to consider that R_1 and R_2 do not contain simple tuples in $\Sigma \times \mathcal{P}(T)$, but instead contain tuples of classes. Within our implementation the intersection of character classes is calculated, similar to code Fragment 9.

```
characterTransitions = {
    <statePair(s1, s2), character(charClass, merge(tags1, tags2)), statePair(s1New,
     ↪  s2New)>
    | <s1, character(charClass1, tags1), s1New> <- R1, <s2, character(charClass2,
     ↪  tags2), s2New> <- R2,
    charClass := intersection(charClass1, charClass2) && size(charClass)>0
}

TagsClass merge(TagsClass tc1, TagsClass tc2) = {tagSet1 + tagSet2 | tagSet1 <-
 ↪  tc1, tagSet2 <- tc2};
```

Fragment 9: Character Transition Classes

The actual implementation still operates slightly differently however, since it only calculates reachable states and their transitions in order to prevent performing unnecessary work. This is done by starting at the initial state, and performing a search from there. For every state in the queue, we iteratively remove it from the queue, calculate its transitions to add to the output, and add any newly encountered states to the queue.

This also introduces some complexity when calculating the TCDFA version of a given TCNFA, since we now can have overlap between transition conditions, without them being fully equivalent. Consider for instance the following simplified transitions (without tag classes):

$$s_1 \xrightarrow{[a-z]} s_2$$
$$s_1 \xrightarrow{[a-k]} s_3$$

When calculating the TCDFA, when encountering state $\{s_1\}$ we can not simply create the transition $\{s_1\} \xrightarrow{[a-z]} \{s_2, s_3\}$, since the characters l to z can not transition to state s_3. In order to deal with this, we first want to calculate the disjoint character classes that make up all character classes in the input. Given a set of character classes C we calculate the set of disjoint character classes D, such that:

$$(\cup_{c \in C} c) = (\cup_{d \in D} d) \tag{1}$$
$$\wedge \forall\, d_1, d_2 \in D \,.\, d_1 \neq d_2 \implies d_1 \cap d_2 = \emptyset \tag{2}$$

Statement 1 specifies that the union of character classes of C and D are equivalent, allowing for all the same transitions. Statement 2 specifies that the character classes of D are indeed disjoint. We can then create a transition for each of these disjoint character classes for our TCDFA, and in the output set include any states that has a transition whose class overlaps this disjoint class (*i.e.* has a non-empty intersection). For our previous example, our input set would be $C = \{[a-z], [a-k]\}$, and the disjoint output set would be $D = \{[a-k], [l-z]\}$. The resulting output transitions for $\{s_1\}$ would be $\{s_1\} \xrightarrow{[a-k]} \{s_2, s_3\}$ and $\{s_1\} \xrightarrow{[l-z]} \{s_2\}$. While calculating the disjoint character classes we can immediately calculate the overlapping character classes from the input, to prevent additional intersection calculations later.

The algorithm that calculates these disjoint character classes is based on a sweep-line approach as commonly seen in geometric algorithms. The character ranges of characters classes are interpreted as 1 dimensional geometric ranges. For each range in a character class, we generate an event at the start of the range which indicates that the class became active, and an event at the end of the range which indicates that the class became inactive. This ending index is one greater than the index specified in each of the character ranges, because the character range treats this last index as inclusive. Next, all events for a given index are merged together, storing which classes become active and which become inactive per index, and then sorted to be in increasing ordering. Now we can define a set of active classes $A := \emptyset$, a previous range index $i_p := -1$, and a list of pairs of character-ranges and sets of character classes $o := []$. The events are then iterated over in increasing order. For each event at index i, we first create a copy of the previous active

classes. We then calculate the new active classes by subtracting the classes that the event indicated to have become inactive, and adding the events that the event indicated to have become active. If the newly active classes are identical to the old ones, we skip to the next event. Otherwise, if the old set of active character classes is non-empty, we add the pair $([i_p{-}i], A)$ to our list o. Finally we set $i_p := i$, before moving onto the next event. After processing all events, we are left with a list of character ranges, together with all classes that are active per character range. By ensuring the old active classes to be different from the new active classes, we prevent getting classes such as $[a{-}de{-}z]$ in our output, which are now automatically simplified to $[a{-}z]$. Finally we use our list of active class set per range, to calculate the set of active ranges per active class set, by combining active ranges with equivalent active class sets. This set of active ranges then forms one of the disjoint character classes of the output, and is paired with all active character classes of the input that intersect it. Fragment 10 contains the complete code for this function.

```
data CharClassRegion = ccr(CharClass cc, set[CharClass] includes);
alias Event = tuple[int char, set[CharClass] endClasses, set[CharClass]
 ↪  beginClasses];

set[CharClassRegion] getDisjointCharClasses(set[CharClass] inClasses) {
    set[CharClassRegion] outClasses = {};

    // Setup the events: starting and stopping character classes
    map[int, tuple[set[CharClass] endClasses, set[CharClass] beginClasses]]
     ↪  eventsMap = ();
    void addEvent(int char, set[CharClass] endClasses, set[CharClass]
     ↪  beginClasses) {
        if(char notin eventsMap) eventsMap[char] = <{}, {}>;
        eventsMap[char].endClasses += endClasses;
        eventsMap[char].beginClasses += beginClasses;
    }
    for(class <- inClasses, range <- class) {
        addEvent(range.begin, {}, {class});
        addEvent(range.end + 1, {class}, {}); // The class ends in the next
         ↪  iteration after end, it's still active during end
    }

    list[Event] events = [
        <char, eventsMap[char].endClasses, eventsMap[char].beginClasses>
        | char <- eventsMap];
    events = sort(events, bool (Event a, Event b) { return a.char < b.char; });

    // Go through the events, and keep track of the active classes at any point
    list[tuple[CharRange, set[CharClass]]] setsPerRange = [];
    set[CharClass] activeClasses = {};
    int previousChar = 0;
    for(<char, endClasses, beginClasses> <- events) {
        previousActiveClasses = activeClasses;
        activeClasses -= endClasses;
        activeClasses += beginClasses;
        if(previousActiveClasses == activeClasses) continue; // May happen if
         ↪  ranges weren't fully merged

        if(previousActiveClasses != {})
            setsPerRange += <range(previousChar, char-1), previousActiveClasses>;

        previousChar = char;
    }

    // Combine character classes for all ranges with the same active class set
    map[set[CharClass], list[CharRange]] rangesPerClassSet = ();
    for(<charRange, classSet> <- setsPerRange) {
        if(classSet notin rangesPerClassSet) rangesPerClassSet[classSet] = [];
        rangesPerClassSet[classSet] += charRange;
    }

    return {
        ccr(rangesPerClassSet[charClassSet], charClassSet)
        | charClassSet <- rangesPerClassSet
    };
}
```

Fragment 10: Disjoint Character Classes Calculation

When calculating the transitions for the TCDFA, we also have to consider the disjoint tag classes. For each individual disjoint character class, we also generate the disjoint tags classes. Because there is only a small variety of tags in our application, but no assumptions are made on what tags are exactly, the algorithm for generating these is not as clever or efficient. Fragment 11. This approach also works for character classes, since it assumes nothing on the data domain other than being able to take intersections and differences between two classes. It is however less efficient than the geometric approach for character classes.

```
data TagsClassRegion = tcr(TagsClass tc, set[TagsClass] includes);

set[TagsClassRegion] getDisjointTagsClasses(set[TagsClass] inClasses) {
    set[TagsClassRegion] outClasses = {};
    for(inClass <- inClasses) {
        inClassRemainder = inClass;
        set[TagsClassRegion] newOutClasses = {};
        for(tcr(outClass, parts) <- outClasses) {
            classIntersection = intersection(inClassRemainder, outClass);
            if(isEmpty(classIntersection)) {
                newOutClasses += tcr(outClass, parts);
            } else {
                outClass = subtract(outClass, classIntersection);
                if(!isEmpty(outClass)) newOutClasses += tcr(outClass, parts);
                newOutClasses += tcr(classIntersection, parts + {inClass});

                inClassRemainder = subtract(inClassRemainder, classIntersection);
            }
        }

        if(!isEmpty(inClassRemainder)) newOutClasses += tcr(inClassRemainder,
        ↪  {inClass});
        outClasses = newOutClasses;
    }
    return outClasses;
}
```

Fragment 11: Disjoint Tags Classes Calculation

For TCDFA creation, we need to ensure totality of the transition function as well. To achieve this, we need to calculate transitions that when combined with the existing transitions account for all possible inputs. This can be done quite easily by consider the entire character class that represents the universe of all characters, and subtracting the character classes of transitions we already have. This remainder of the universe can then be paired with the universe of all tag sets. This universe of tag sets is provided as a parameter when creating the TCDFA. For other transitions, we also calculate the complement of their tags class, by subtracting it from the tags set universe. If this complement is non-empty, we also define a transition for the character-class paired with this tags complement.

Finally we would like to be able to fully normalize our TCNFAs. The normalization function N should have the following properties:

1. $\forall\, n_1, n_2 \in TCNFA(\Sigma, T)\ .\ \mathcal{L}(n_1) = \mathcal{L}(n_2) \implies N(n_1) = N(n_2)$
2. $\forall\, n \in TCNFA(\Sigma, T)\ .\ \mathcal{L}(n) = \mathcal{L}(N(n))$

Then using the second property we can also conclude that for any two TCNFAs n_1 and n_2:

$$\begin{aligned} & N(n_1) = N(n_2) \\ \implies & \mathcal{L}(N(n_1)) = \mathcal{L}(N(n_2)) \\ \implies & \mathcal{L}(n_1) = \mathcal{L}(n_2) \end{aligned}$$

Hence $\forall\ n_1, n_2 \in TCNFA(\Sigma, T)\ .\ \mathcal{L}(n_1) = \mathcal{L}(n_2) \iff N(n_1) = N(n_2)$. Therefore we can check for language equivalence, by checking for TCNFA equivalence. This is useful because languages are likely infinite, while our TCNFAs are merely finite. This allows us to use all of Rascal's features that make use of equivalence checks too, like using TCNFAs as keys in maps, or entries in sets. This way we do not have to care about syntactic differences in regular expressions, and can directly reason about the languages they define.

Minimization of DFAs is a well-researched topic, for which multiple algorithms have already been defined. For every DFA, a unique minimum DFA – modulo state naming – exists with an equivalent language. This makes the DFA minimization procedure ideal for our normalization purposes, since we already know how to generate a TCDFA counterpart from a TCNFA. Just like in the process of converting a TCNFA to a TCDFA, we can ignore the state separation when performing the normalization, and extract it again afterwards. Moreover, in reality we do not actually explicitly store the state partition or set in our implementation. We instead calculate it on the fly based on the initial state and existing transitions when we need the information about this partition. After we minimize the given TCDFA we remove impotent transitions. These are transitions that do not contribute to phrases in the language. This will get rid of any states that would not fit a single set of the partition. And for our purposes, the automaton need not be complete anyhow. We then also need to relabel all states, in a predictable way. We label the states using numeric integer identifiers that increase for every state. We know two minimized TCDFAs with the same language to be isomorphic, so if we label the initial state as 0, and then perform a search from the initial state, all other states should be reached in the same order, if we can iterate over out-going transitions in a predictable way. Rascal already allows any collection of values to be ordered based on the $<$ operator defined on the value type. This $<$ is automatically defined for any custom data type, using an ordering on the constructors and a lexicographical ordering on constructor arguments. The only problem is that $<$ does not provide a complete ordering on list and set values. We used lists for defining scopes, which we use as tags in regular expressions. This would prevent proper predictable sorting. Therefore we do not use lists here, but instead use a custom data type:

```
data Scopes = noScopes()
            | someScopes(Scope top, Scopes bottom);
```

This allows us to define sortable linked lists using this custom data type. Sets are used on the top level in order to define our tags classes. Therefore we assume individual tags to be sortable using rascals $<$ operation, and manually take care of sorting the sets on the top-level. In general, we do not actually care how values are sorted, as long as the sorting is table. We want to rely on the values of data for sorting, not arbitrary hashes or other unstable things in the programming language. Therefore we just have to define any scheme that given two sets of values that are not equivalent, specifies which of them is smaller. We can for instance do this by retrieving all values that are not shared between the two sets, and obtaining the smallest value among these based on a recursive sorting criterion (using either this set comparison scheme and Rascal's $<$ operator), and deem the set to which this value belongs to be the smallest set. Using these mechanisms we can consistently sort any list of transitions, and therefore consistently iterate through all states of a minimized TCDFA, such that labeling according to this ordering results in equivalent automata given TCDFAs with an equivalent language.

We are however left with one more problem caused by our use of character and tags classes. The same automaton can be represented in multiple different ways by either splitting a transition into multiple transitions by splitting one of its classes. Because of this, we have to take care of two additional things:

- Make all transitions of the TCDFA fully disjoint, to ensure maximal minimization
- Normalize the transitions of the resulting minimal TCDFA

To see that non-disjointness could pose problems, we have to consider how the minimization algorithm operates. This algorithm is based on the code assuming all states to belong to the same equivalence class, until we can show some state not to belong to this class. The first step in this process is separating the class of accepting and non-accepting states, since these are trivially different. After this, states are separated based on the transitions they have to different equivalence classes. Transitions with character classes that overlap but are not identical can form problems here, because two states in the same equivalence class that specify the same behavior, may suddenly no longer have the same edges to another equivalence class. The minimization algorithm could be modified to consider this potential overlap, but this might prove complex. A simpler approach is to ensure edges never overlap unless they are equal. This can be achieved by taking all transition conditions together, and calculating the disjoint set of transition conditions, just like we did for the DFA conversion process. Then any edge can be replaced by a combination of edges using the disjoint transition conditions that together combine to the condition of the original edge. This same approach could also have been taken for our regex to TCNFA conversion algorithm, such that we do not have to deal with partial overlap there, but this was not done because the complexity of the conversion code was not significantly increased by the presence of character and tags classes.

The edge normalization step afterwards is still required to ensure that resulting minimal TCDFAs are not only minimal in number of states, but also in number of edges to ensure equivalence. For DFAs that do not use character classes, these conditions of minimal in number of states and in number of edges happen to be equivalent, because the automata are complete. Our edge normalization consists of achieving the following conditions:

1. $\forall\ u \xrightarrow{(CC,TC)} v\ .\ |CC| > 0 \wedge |TC| > 0$, *i.e.* every transition must accept some some input
2. $\forall\ u \xrightarrow{(CC,TC)} v, u \xrightarrow{(CC',TC')} v\ .\ TC \cap TC' = \emptyset$, *i.e.* for every pair of transitions between the same states, no tags are shared
3. $\forall\ u \xrightarrow{(CC,TC)} v, u \xrightarrow{(CC',TC')} v\ .\ CC \neq CC'$, *i.e.* for every pair of transitions between the same states, their character classes are not equivalent

For each of these constraints, if they are not met, we can get rid of redundancy in the automaton:

1. If there is an edge that cannot be taken, we might as well remove the edge
2. If two transitions share some tags, the character classes of the shared tags could have been merged. Note that We arbitrary decided to prioritize character class merging over tags class merging
3. If two transitions have the same character classes, we could replace them by a single transition with this character class and the union of tags classes

Our goal with these constraints is to find a set of normalized transitions between two states, given the current set of transitions between states. Let CCU be the universe of character classes and TCU be the universe of tag classes. Then a transition condition c is a pair in $CCU \times TCU$. We define the language $\mathcal{L}(S)$ for a set $S \subset CCU \times TCU$ of transition conditions, to be the set of all character and tagset pairs that are accepted by a transition condition in S. Now our transition normalization function N should have the following properties to ensure our desired normalization purpose:

1. $\forall\ s_1, s_2 \subseteq CCU \times TCU\ .\ \mathcal{L}(s_1) = \mathcal{L}(s_2) \implies N(s_1) = N(s_2)$
2. $\forall\ s \subseteq CCU \times TCU\ .\ \mathcal{L}(s) = \mathcal{L}(N(s))$

It can be proven that this first condition follows from our 3 constraints. Achieving the constraints is trivial:

1. Remove transitions with empty character classes or tags classes
2. Make all tags classes disjoint by calculating the disjoint tags classes and splitting transition conditions accordingly
3. Per disjoint tags class, union all corresponding character classes into one condition

4. For transition conditions with the same character classes, union their tags classes together

Step 1 trivially achieves meeting constraint 1, and no later unioning can invalidate this constraint. Constraint 2 is met by step 2, and is maintained by step 3 and 4. It is maintained because unioning character classes will not introduce intersection between tags classes. And after all character classes have been unioned, there will only be a single transition constraint per tags class. Therefore step 4 won't be able to union two transitions sets together with overlapping tags classes. Finally this 4th step trivially achieves our third constraint.

In summary, our normalization ensures that a normalized TCNFA uniquely represents its language, and this is achieved using the following steps:

1. Transform our TCNFA to a TCDFA
2. Remove partial overlap between transitions by splitting overlapping transitions
3. Apply a DFA minimization algorithm
4. Normalize the edges between every pair of states
5. Relabel TCDFA states according to ordering on transitions

C Regular Expression Subtraction Replacement Function

We inductively define function $S : \mathcal{R}(\Sigma, T) \to \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathbb{B}$ that splits a regular expression in 4 parts, and a boolean specifying whether equivalence is maintained. This function relies on a simple helper function $S_l : \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \times \mathcal{R}(\Sigma, T) \to \mathcal{R}(\Sigma, T) \times \mathbb{B}$ which attempts to combine the four expressions b, m, a and s into an expression y either as $y = /(\not> bsa)m/$ or $y = /m(bsa\not<)/$. This function attempts to ensure that $\mathcal{L}(y) = \mathcal{L}(m{-}s)$, and returns a boolean indicating whether this was achieved.

Using this, we can inductively define S, where we use $(b_1, m_1, a_1, s_1, e_1) = S(r_1)$ and $(b_2, m_2, a_2, s_2, e_2) = S(r_2)$ as the values obtained from sub-expressions:

$$
\begin{aligned}
S(\mathbf{0}) &= (\epsilon, \mathbf{0}, \epsilon, \mathbf{0}, \mathit{true}) \\
S(\mathbf{1}) &= (\epsilon, \mathbf{1}, \epsilon, \mathbf{0}, \mathit{true}) \\
S(\epsilon) &= (\epsilon, \epsilon, \epsilon, \mathbf{0}, \mathit{true}) \\
S(a) &= (\epsilon, a, \epsilon, \mathbf{0}, \mathit{true}) \\
S(r_1{+}r_2) &= (\epsilon, m_1'{+}m_2', \epsilon, \mathbf{0}, e_1 \wedge e_2 \wedge e_1' \wedge e_2') && \text{where } (m_1', e_1') = S_l(b_1, m_1, a_1, s_1), (m_2', e_2') = S_l(b_2, m_2, a_2, s_2) \\
S(r_1 r_2) &= (b_1, m_1 m_2, a_2, s_1 m_2{+}m_1 s_2, e_1 \wedge e_2) \\
S(r_1^+) &= (\epsilon, m^+, \epsilon, \mathbf{0}, e_1 \wedge e) && \text{where } (m, e) = S_l(b_1, m_1, a_1, s_) \\
S(r_1{>}r_2) &= (b_1, m_1{>}a', a_1{>}a', s_1, e_1 \wedge e_2 \wedge e) && \text{where } (a', e) = S_l(b_2, m_2, a_2, s_2) \\
S(r_2{<}r_1) &= (b'{<}b_1, a'{<}m_1, a_1, s_1, e_1 \wedge e_2 \wedge e) && \text{where } (b', e) = S_l(b_2, m_2, a_2, s_2) \\
S(r_1{\not>}r_2) &= (b_1, m_1{\not>}a', a_1{\not>}a', s_1, e_1 \wedge e_2 \wedge e) && \text{where } (a', e) = S_l(b_2, m_2, a_2, s_2) \\
S(r_2{\not<}r_1) &= (b'{\not<}b_1, b'{\not<}m_1, a_1, s_1, e_1 \wedge e_2 \wedge e) && \text{where } (b', e) = S_l(b_2, m_2, a_2, s_2) \\
S(r_1{-}r_2) &= (b_1, m_1, a_1, s_1{+}s', e_1 \wedge e_2 \wedge e) && \text{where } (s', e) = S_l(b_2, m_2, a_2, s_2) \\
S((\langle t \rangle r_1)) &= (b_1, (\langle t \rangle m_1), a_1, s_1, e_1)
\end{aligned}
$$

D Regular Expression Scope Extraction

We define a recursive function E that operates on a regular expression and a category list and outputs a new regular expression only containing numbered capture groups together with a list of categories corresponding to every capture group.

$$
\begin{aligned}
E(\mathbf{0},\ s) &= (\mathbf{0},\ s) \\
E(\mathbf{1},\ s) &= (\mathbf{1},\ s) \\
E(\epsilon,\ s) &= (\epsilon,\ s) \\
E(a,\ s) &= (a,\ s) \\
E(r_1{+}r_2,\ s) &= (r_1'{+}r_2',\ s'') && \text{where } (r_1', s') = E(r_1, s) \wedge (r_2', s'') = E(r_2, s') \\
E(r_1 r_2,\ s) &= (r_1' r_2',\ s'') && \text{where } (r_1', s') = E(r_1, s) \wedge (r_2', s') = E(r_2, s') \\
E(r_1^+,\ s) &= (r_1'^+,\ s'') && \text{where } (r_1', s') = E(r_1, s) \\
E(r_1{>}r_2,\ s) &= (r_1'{>}r_2',\ s'') && \text{where } (r_1', s') = E(r_1, s) \wedge (r_2', s'') = E(r_2, s') \\
E(r_1{\not>}r_2,\ s) &= (r_1'{\not>}r_2',\ s'') && \text{where } (r_1', s') = E(r_1, s) \wedge (r_2', s'') = E(r_2, s') \\
E(r_2{<}r_1,\ s) &= (r_2'{<}r_1',\ s'') && \text{where } (r_1', s') = E(r_1, s) \wedge (r_2', s'') = E(r_2, s') \\
E(r_2{\not<}r_1,\ s) &= (r_2'{\not<}r_1',\ s'') && \text{where } (r_1', s') = E(r_1, s) \wedge (r_2', s'') = E(r_2, s') \\
E(r_1{-}r_2,\ s) &= (r_1'{-}r_2',\ s'') && \text{where } (r_1', s') = E(r_1, s) \wedge (r_2', s'') = E(r_2, s') \\
E((\langle t \rangle r_1), s) &= ((\langle |st| \rangle r_1'), s') && \text{where } (r_1', s') = E(r_1, st)
\end{aligned}
$$

E Regular Expression Tags To Leaves

We define recursive function M that uses a chosen *merge* function to move all tag to the leaves, such that every regular expression contains exactly one tag. Below is a condensed overview of the function for an input expression x built up from at most the sub-expressions r_1 and r_2:

$$
\begin{aligned}
M(\mathbf{0},\ c) &= \mathbf{0} \\
M(\mathbf{1},\ c) &= (\langle c \rangle \mathbf{1}) \\
M(\epsilon,\ c) &= \epsilon \\
M(a,\ c) &= (\langle c \rangle a) \\
M(r_1{+}r_2,\ c) &= r_1'{+}r_2' && \text{where } r_1' = M(r_1, c) \wedge r_2' = M(r_2, c) \\
M(r_1 r_2,\ c) &= r_1' r_2' && \text{where } r_1' = M(r_1, c) \wedge r_2' = M(r_2, c) \\
M(r_1^+,\ c) &= r_1'^+ && \text{where } r_1' = M(r_1, c) \\
M(r_1{>}r_2,\ c) &= r_1'{>}r_2 && \text{where } r_1' = M(r_1, c) \\
M(r_1{\not>}r_2,\ c) &= r_1'{\not>}r_2 && \text{where } r_1' = M(r_1, c) \\
M(r_2{<}r_1,\ c) &= r_2{<}r_1' && \text{where } r_1' = M(r_1, c) \\
M(r_2{\not<}r_1,\ c) &= r_2{\not<}r_1' && \text{where } r_1' = M(r_1, c) \\
M(r_1{-}r_2,\ c) &= r_1'{-}r_2 && \text{where } r_1' = M(r_1, c) \\
M((\langle t \rangle r_1), c) &= M(r_1, c') && \text{where } c' = \mathit{merge}(t, c)
\end{aligned}
$$

F Regular Expression Tagged Alternation Removal

We define an inductive function $S : \mathcal{R}(\Sigma, T) \to \mathcal{P}(\mathcal{R}(\Sigma, T))$, such that for any regular expression x, we have $\mathcal{L}(x) = \mathcal{L}(/+_{y \in S(x)}\ y/)$. Let $R_1 = S(r_1)$ and $R_2 = S(r_2)$ be the recursively defined results from

sub-expressions, allowing us to define S:

$$\begin{aligned}
S(\mathbf{0}) &= \{\mathbf{0}\} \\
S(\mathbf{1}) &= \{\mathbf{1}\} \\
S(\epsilon) &= \{\epsilon\} \\
S(a) &= \{a\} \\
S(r_1{+}r_2) &= R_1 \cup R_2 \\
S(r_1 r_2) &= \{r_1' r_2' \mid r_1' \in R_1, r_2' \in R_2\} \\
S(r_1^+) &= \{r_1\} \\
S(r_1{>}r_2) &= \{r_1'{>}r_2 \mid r_1' \in R_1\} \\
S(r_1{\not>}r_2) &= \{r_2{\not<}r_1' \mid r_1' \in R_1\} \\
S(r_2{<}r_1) &= \{r_1'{>}r_2 \mid r_1' \in R_1\} \\
S(r_2{\not<}r_1) &= \{r_2{\not<}r_1' \mid r_1' \in R_1\} \\
S(r_1{-}r_2) &= \{r_1'{-}r_2 \mid r_1' \in R_1\} \\
S((\langle t \rangle r_1)) &= \{(\langle t \rangle r_1)\}
\end{aligned}$$

G Regular Expression Equivalence Axioms

We will provide several axioms for regular expression language equivalences that involve lookarounds and or language subtraction. We use the symbol $\equiv$ to denote language equivalence of the provide expressions. Most of the axioms apply regardless of presence of tags, but some axioms only apply for regular expressions without tags. These axioms have been provided later. Proofs for most cases are provided, but these are provided using the language definitions for readability, and the symmetric proves that are nearly identical have been omitted.

- $x{<}(y{>}z) \equiv (x{<}y){>}z$ (Lookaround Associativity)
  - $x{<}(y{\not>}z) \equiv (x{<}y){\not>}z$
  - $x{\not<}(y{>}z) \equiv (x{\not<}y){>}z$
  - $x{\not<}(y{\not>}z) \equiv (x{\not<}y){\not>}z$
  - $(y{>}z){>}x \equiv (y{>}x){>}z$
  - $(y{>}z){\not>}x \equiv (y{\not>}x){>}z$
  - $(y{\not>}z){>}x \equiv (y{>}x){\not>}z$
  - $(y{\not>}z){\not>}x \equiv (y{\not>}x){\not>}z$
  - $x{<}(z{<}y) \equiv z{<}(x{<}y)$
  - $x{\not<}(z{<}y) \equiv z{<}(x{\not<}y)$
  - $x{<}(z{\not<}y) \equiv z{\not<}(x{<}y)$
  - $x{\not<}(z{\not<}y) \equiv z{\not<}(x{\not<}y)$
- $x{>}(y{>}z) \equiv x{>}(yz)$ (Double Lookaround Elimination)
  - $(x{<}y){<}z \equiv (xy){<}z$
- $(x{>}y){-}z \equiv (x{-}z){>}y$ (Lookaround Subtraction Associativity)
  - $(x{\not>}y){-}z \equiv (x{-}z){\not>}y$
  - $(x{<}y){-}z \equiv x{<}(y{-}z)$
  - $(x{\not<}y){-}z \equiv x{\not<}(y{-}z)$

- $x{>}(y{+}z) \equiv (x{>}y){+}(x{>}z)$ (Lookaround Distributivity)
  - $(y{+}z){<}x \equiv (y{<}x){+}(z{<}x)$
- $(x{+}y){>}z \equiv (x{>}z){+}(y{>}z)$ (Lookaround Match Distributivity)
  - $(x{+}y){\not>}z \equiv (x{\not>}z){+}(y{\not>}z)$
  - $z{<}(x{+}y) \equiv (z{<}x){+}(z{<}y)$
  - $z{\not<}(x{+}y) \equiv (z{\not<}x){+}(z{\not<}y)$
  - $(x{-}y){>}z \equiv (x{>}z){-}(y{>}z)$
  - $(x{-}y){\not>}z \equiv (x{\not>}z){-}(y{\not>}z)$
  - $z{<}(x{-}y) \equiv (z{<}x){-}(z{<}y)$
  - $z{\not<}(x{-}y) \equiv (z{\not<}x){-}(z{\not<}y)$
- $(x{-}y){-}z \equiv x{-}(y{+}z)$ (Double Subtraction Elimination)
- $(x{+}y){-}z \equiv (x{-}z){+}(y{-}z)$ (Subtraction Distributivity)
- $x{>}\epsilon \equiv x$ (Infallible Lookaround)
  - $\epsilon{<}x \equiv x$
  - $x{>}\mathbf{1} \equiv x$
  - $\mathbf{1}{<}x \equiv x$
- $x{>}\mathbf{0} \equiv \mathbf{0}$ (Unfulfillable Lookaround)
  - $\mathbf{0}{<}x \equiv \mathbf{0}$
- $x{\not>}\mathbf{0} \equiv x$ (Infallible Negative Lookaround)
  - $\mathbf{0}{\not<}x \equiv x$
- $x{\not>}\epsilon \equiv \mathbf{0}$ (Unfulfillable Negative Lookaround)
  - $\epsilon{\not<}x \equiv \mathbf{0}$
  - $x{\not>}\mathbf{1} \equiv \mathbf{0}$
  - $\mathbf{1}{\not<}x \equiv \mathbf{0}$

Axioms for regular expression not allowing tags:

- $(x{>}y){+}x \equiv x$ (Constraint Weakening)
  - $(y{<}x){+}x \equiv x$
  - $(x{\not>}y){+}x \equiv x$
  - $(y{\not<}x){+}x \equiv x$
  - $(x{-}y){+}x \equiv x$

Axioms for regular expressions requiring tags:

- $(\langle s\rangle(\langle t\rangle x)) \equiv (\langle t\rangle(\langle s\rangle x))$ (Double Tag Associativity)
- $(\langle t\rangle x{>}y) \equiv (\langle t\rangle x){>}y$ (Tag Constraint Associativity)
  - $(\langle t\rangle x{\not>}y) \equiv (\langle t\rangle x){\not>}y$
  - $(\langle t\rangle y{<}x) \equiv y{<}(\langle t\rangle x)$
  - $(\langle t\rangle y{\not<}x) \equiv y{\not<}(\langle t\rangle x)$
  - $(\langle t\rangle x{-}y) \equiv (\langle t\rangle x){-}y$

G.1 Lookaround Associativity Proof

$$
\begin{aligned}
\mathcal{L}(x{<}(y{>}z)) &= \{(p,w,s) \in \mathcal{L}(y{>}z) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha,\beta,ws) \in \mathcal{L}(x)\} \\
&= \{(p,w,s) \in \{(p',w',s') \in \mathcal{L}(y) \mid \exists\, \alpha',\beta' \in \Sigma^* \,.\, s' = \alpha'\beta' \wedge (p'w',\alpha',\beta') \in \mathcal{L}(z)\} \\
&\quad \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha,\beta,ws) \in \mathcal{L}(x)\} \\
&= \{(p,w,s) \in \mathcal{L}(y) \mid \exists\, \alpha',\beta' \in \Sigma^* \,.\, s = \alpha'\beta' \wedge (pw,\alpha',\beta') \in \mathcal{L}(z) \\
&\quad \wedge \exists\, \alpha,\beta \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha,\beta,ws) \in \mathcal{L}(x)\} \\
&= \{(p,w,s) \in \mathcal{L}(y) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha,\beta,ws) \in \mathcal{L}(x) \\
&\quad \wedge \exists\, \alpha',\beta' \in \Sigma^* \,.\, s = \alpha'\beta' \wedge (pw,\alpha',\beta') \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \{(p',w',s') \in \mathcal{L}(y) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s' = \alpha\beta \wedge (p'w',\alpha',\beta') \in \mathcal{L}(x)\} \\
&\quad \mid \exists\, \alpha',\beta' \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha',\beta',ws) \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x{<}y) \mid \exists\, \alpha',\beta' \in \Sigma^* \,.\, p = \alpha\beta \wedge (\alpha',\beta',ws) \in \mathcal{L}(z)\} \\
&= \mathcal{L}((x{<}y){>}z)
\end{aligned}
$$

G.2 Double Lookaround Elimination Proof

$$
\begin{aligned}
\mathcal{L}(x{>}(y{>}z)) &= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y{>}z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \{ \\
&\quad (p',w',s') \in \mathcal{L}(y) \mid \exists\, \alpha',\beta' \in \Sigma^* \,.\, s' = \alpha'\beta' \wedge (p'w',\alpha',\beta') \in \mathcal{L}(z)\}\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \\
&\quad \wedge (pw,\alpha,\beta) \in \mathcal{L}(y) \wedge (\exists\, \alpha',\beta' \in \Sigma^* \,.\, \beta = \alpha'\beta' \wedge (pw\alpha,\alpha',\beta') \in \mathcal{L}(z))\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, \exists\, \alpha',\beta' \in \Sigma^* \,. \\
&\quad s = \alpha\beta \wedge \beta = \alpha'\beta' \wedge (pw,\alpha,\beta) \in \mathcal{L}(y) \wedge (pw\alpha,\alpha',\beta') \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\alpha',\beta' \in \Sigma^* \,.\, s = \alpha\alpha'\beta' \\
&\quad \wedge (pw,\alpha,\alpha'\beta') \in \mathcal{L}(y) \wedge (pw\alpha,\alpha',\beta') \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \gamma,\beta' \in \Sigma^* \,.\, \exists\, \alpha,\alpha' \in \Sigma^* \,. \\
&\quad \gamma = \alpha\alpha' \wedge s = \gamma\beta' \wedge (pw,\alpha,\alpha'\beta') \in \mathcal{L}(y) \wedge (pw\alpha,\alpha',\beta') \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \gamma,\beta' \in \Sigma^* \,.\, s = \gamma\beta' \wedge \exists\, \alpha,\alpha' \in \Sigma^* \,. \\
&\quad \gamma = \alpha\alpha' \wedge (pw,\alpha,\alpha'\beta') \in \mathcal{L}(y) \wedge (pw\alpha,\alpha',\beta') \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \gamma,\beta' \in \Sigma^* \,.\, s = \gamma\beta' \wedge (pw,\gamma,\beta') \in \{ \\
&\quad (p',w',s') \mid \exists\, \alpha,\alpha' \in \Sigma^* \,.\, w' = \alpha\alpha' \\
&\qquad \wedge (p',\alpha,\alpha's') \in \mathcal{L}(y) \wedge (p'\alpha,\alpha',s') \in \mathcal{L}(z)\}\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \gamma,\beta' \in \Sigma^* \,.\, s = \gamma\beta' \wedge (pw,\gamma,\beta') \in \{ \\
&\quad (p',\alpha\alpha',s') \mid (p',\alpha,\alpha's') \in \mathcal{L}(y) \wedge (p'\alpha,\alpha',s') \in \mathcal{L}(z)\}\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \gamma,\beta' \in \Sigma^* \,.\, s = \gamma\beta' \wedge (pw,\gamma,\beta') \in \mathcal{L}(yz)\} \\
&= \mathcal{L}(x{>}(yz))
\end{aligned}
$$

G.3 Lookaround Subtraction Associativity Proof

$$\begin{aligned}
\mathcal{L}((x{>}y){-}z) &= \mathcal{L}(x{>}y)\backslash\mathcal{L}(z) \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)\}\backslash\mathcal{L}(z) \\
&= \{(p,w,s) \in \mathcal{L}(x)\backslash\mathcal{L}(z) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)\} \\
&= \{(p,w,s) \in \mathcal{L}(x{-}z) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)\} \\
&= \mathcal{L}((x{-}z){>}y)
\end{aligned}$$

G.4 Lookaround Distributivity Proof

$$\begin{aligned}
\mathcal{L}(x{>}(y{+}z)) &= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y{+}z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y) \cup \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge ((pw,\alpha,\beta) \in \mathcal{L}(y) \vee (pw,\alpha,\beta) \in \mathcal{L}(z))\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid (\exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)) \\
&\qquad \vee (\exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(z))\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)\} \\
&\qquad \cup \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(z)\} \\
&= \mathcal{L}(x{>}y) \cup \mathcal{L}(x{>}z) \\
&= \mathcal{L}((x{>}y){+}(x{>}z))
\end{aligned}$$

G.5 Lookaround Match Distributivity Proof

$$\begin{aligned}
\mathcal{L}((x{+}y){>}z) &= \{(p,w,s) \in \mathcal{L}(x+y) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \cup \mathcal{L}(y) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(z)\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(z)\} \\
&\qquad \cup \{(p,w,s) \in \mathcal{L}(y) \mid \exists\, \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(z)\} \\
&= \mathcal{L}(x{>}z) \cup \mathcal{L}(y{>}z) \\
&= \mathcal{L}((x{>}z){+}(y{>}z))
\end{aligned}$$

G.6 Double Subtraction Elimination Proof

$$\begin{aligned}
\mathcal{L}((x{-}y){-}z) &= \mathcal{L}(x{-}y)\backslash\mathcal{L}(z) \\
&= (\mathcal{L}(x)\backslash\mathcal{L}(y))\backslash\mathcal{L}(z) \\
&= \mathcal{L}(x)\backslash(\mathcal{L}(y) \cup \mathcal{L}(z)) \\
&= \mathcal{L}(x)\backslash\mathcal{L}(y{+}z) \\
&= \mathcal{L}(x{-}(y{+}z))
\end{aligned}$$

G.7 Subtraction Distributivity Proof

$$\begin{aligned}\mathcal{L}((x{+}y){-}z) &= \mathcal{L}(x{+}y)\backslash\mathcal{L}(z)\\ &= (\mathcal{L}(x)\cup\mathcal{L}(y))\backslash\mathcal{L}(z)\\ &= (\mathcal{L}(x)\backslash\mathcal{L}(z))\cup(\mathcal{L}(y)\backslash\mathcal{L}(z))\\ &= \mathcal{L}(x{-}z)\cup\mathcal{L}(y{-}z)\\ &= \mathcal{L}((x{-}z){+}(y{-}z))\end{aligned}$$

G.8 Infallible Lookaround Proof

$$\begin{aligned}\mathcal{L}(x{>}\epsilon) &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge(pw,\alpha,\beta)\in\mathcal{L}(\epsilon)\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge(pw,\alpha,\beta)\in\{(p',\epsilon,s')\mid p',s'\in\Sigma^*\}\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge pw,s\in\Sigma^*\wedge\alpha=\epsilon\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge\mathit{true}\wedge\alpha=\epsilon\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\beta\in\Sigma^*\,.\,s=\epsilon\beta\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\mathit{true}\}\\ &= \mathcal{L}(x)\end{aligned}$$

G.9 Unfulfillable Lookaround Proof

$$\begin{aligned}\mathcal{L}(x{>}\mathbf{0}) &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge(pw,\alpha,\beta)\in\mathcal{L}(\mathbf{0})\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge(pw,\alpha,\beta)\in\emptyset\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge\mathit{false}\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\mathit{false}\}\\ &= \emptyset\\ &= \mathcal{L}(\mathbf{0})\end{aligned}$$

G.10 Unfulfillable Negative Lookaround Proof

$$\begin{aligned}\mathcal{L}(x{\not>}\epsilon) &= \{(p,w,s)\in\mathcal{L}(x)\mid\neg\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge(pw,\alpha,\beta)\in\mathcal{L}(\epsilon)\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\neg\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge(pw,\alpha,\beta)\in\{(p',\epsilon,s')\mid p',s'\in\Sigma^*\}\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\neg\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge pw,s\in\Sigma^*\wedge\alpha=\epsilon\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\neg\exists\,\alpha,\beta\in\Sigma^*\,.\,s=\alpha\beta\wedge\mathit{true}\wedge\alpha=\epsilon\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\neg\exists\,\beta\in\Sigma^*\,.\,s=\epsilon\beta\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\neg\mathit{true}\}\\ &= \{(p,w,s)\in\mathcal{L}(x)\mid\mathit{false}\}\\ &= \emptyset\\ &= \mathcal{L}(\mathbf{0})\end{aligned}$$

G.11 Infallible Negative Lookaround Proof

$$\begin{aligned}
\mathcal{L}(x{\not>}\mathbf{0}) &= \{(p,w,s) \in \mathcal{L}(x) \mid \neg\exists\ \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(\mathbf{0})\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \neg\exists\ \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \emptyset\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \neg\exists\ \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge \mathit{false}\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \neg\mathit{false}\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \mathit{true}\} \\
&= \mathcal{L}(x)
\end{aligned}$$

G.12 Constraint Weakening Proof

$$\begin{aligned}
\mathcal{L}((x{>}y){+}x) &= \mathcal{L}(x{>}y) \cup \mathcal{L}(x) \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\ \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)\} \cup \mathcal{L}(x) \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \exists\ \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)\} \\
&\quad \cup \{(p,w,s) \in \mathcal{L}(x) \mid \mathit{true}\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid (\exists\ \alpha,\beta \in \Sigma^* \,.\, s = \alpha\beta \wedge (pw,\alpha,\beta) \in \mathcal{L}(y)) \vee \mathit{true}\} \\
&= \{(p,w,s) \in \mathcal{L}(x) \mid \mathit{true}\} \\
&= \mathcal{L}(x)
\end{aligned}$$

G.13 Double Tag Associativity Proof

$$\begin{aligned}
\mathcal{L}((\langle t\rangle(\langle t'\rangle x))) &= \{(p,(w,\{t\} \cup T^w),s) \mid (p,(w,T^w),s) \in \mathcal{L}((\langle t'\rangle x))\} \\
&= \{(p,(w,\{t\} \cup T^w),s) \mid (p,(w,T^w),s) \in \{ \\
&\quad (p',(w',\{t'\} \cup T^{w'}),s') \mid (p',(w',T^{w'}),s') \in \mathcal{L}(x)\}\} \\
&= \{(p,(w,\{t\} \cup \{t'\} \cup T^w),s) \mid (p,(w,T^{w'}),s) \in \mathcal{L}(x)\} \\
&= \{(p,(w,\{t'\} \cup \{t\} \cup T^w),s) \mid (p,(w,T^{w'}),s) \in \mathcal{L}(x)\} \\
&= \{(p,(w,\{t'\} \cup T^w),s) \mid (p,(w,T^w),s) \in \{ \\
&\quad (p',(w',\{t\} \cup T^{w'}),s') \mid (p',(w',T^{w'}),s') \in \mathcal{L}(x)\}\} \\
&= \{(p,(w,\{t'\} \cup T^w),s) \mid (p,(w,T^w),s) \in \mathcal{L}((\langle t\rangle x))\} \\
&= \mathcal{L}((\langle t'\rangle(\langle t\rangle x)))
\end{aligned}$$

G.14 Tag Constraint Associativity Proof

$$
\begin{aligned}
\mathcal{L}((\langle t\rangle x{>}y)) &= \{(p,(w,\{t\}\cup T^w),s) \mid (p,(w,T^w),s)\in\mathcal{L}(x{>}y)\}\\
&= \{(p,(w,\{t\}\cup T^w),s) \mid (p,(w,T^w),s)\in\{\\
&\quad (((p',T_1^p\cup T_2^p),(w',T_1^w\cup T_2^w),(\alpha\beta,T^s\cup(T^\alpha T^\beta)))\\
&\quad \mid ((p',T_1^p),(w',T_1^w),(\alpha\beta,T^s))\in\mathcal{L}(x)\wedge((p',T_2^p)(w',T_2^w),(\alpha,T^\alpha),(\beta,T^\beta))\in\mathcal{L}(y)\}\}\\
&= \{(((p,T_1^p\cup T_2^p),(w,\{t\}\cup T_1^w\cup T_2^w),(\alpha\beta,T^s\cup(T^\alpha T^\beta)))\\
&\quad \mid ((p,T_1^p),(w,T_1^w),(\alpha\beta,T^s))\in\mathcal{L}(x)\wedge((p,T_2^p)(w,T_2^w),(\alpha,T^\alpha),(\beta,T^\beta))\in\mathcal{L}(y)\}\\
&= \{(((p,T_1^p\cup T_2^p),(w,T_1^w\cup T_2^w),(\alpha\beta,T^s\cup(T^\alpha T^\beta)))\\
&\quad \mid ((p,T_1^p),(w,T_1^w),(\alpha\beta,T^s))\in\{(p',(w',\{t\}\cup T^{w'}),s') \mid (p',(w',T^{w'}),s')\in\mathcal{L}(x)\}\\
&\quad \wedge((p,T_2^p)(w,T_2^w),(\alpha,T^\alpha),(\beta,T^\beta))\in\mathcal{L}(y)\}\\
&= \{(((p,T_1^p\cup T_2^p),(w,T_1^w\cup T_2^w),(\alpha\beta,T^s\cup(T^\alpha T^\beta)))\\
&\quad \mid ((p,T_1^p),(w,T_1^w),(\alpha\beta,T^s))\in\mathcal{L}((\langle t\rangle x))\wedge((p,T_2^p)(w,T_2^w),(\alpha,T^\alpha),(\beta,T^\beta))\in\mathcal{L}(y)\}\\
&= \mathcal{L}((\langle t\rangle x){>}y)
\end{aligned}
$$