UNIVERSITY OF COPENHAGEN

Bachelor's Project

Christian Bendix Fjordstrøm

Hybrid Syntax For Composition of Regular Languages

Date: May 28, 2024

Advisor: Andrzej Filinski

Abstract

Table of content

1 Introduction

2 Analysis

This project revolves around designing and implementing a tool that allows users to input a hybrid syntax of standard regular expressions extended with complement and intersection, regular grammars, and automata. The requirements for the project can thus be broken down into the following parts:

2.1 Hybrid Syntax

The hybrid syntax should allow for arbitrarily nested compositional expressions consisting of regular expressions, regular grammars and automata. Since it is undecidable if a CFG describes a regular language it needs to be restricted. To ensure that the given grammar/automata are regular and can be converted, it must be checked before conversion that the given grammar/automata are well-formed.

2.2 I/O

The user should also be able to define which output format they want, and the syntax of the output should be the same as the syntax for the input, such that it can be fed back in. The output should also exhibit a roundtrip property, meaning that feeding the output back in should result in the same language. Additionally, there should also be an option for a user to check if a string is recognised by the given regular language.

2.3 Composition

The standard regular expression operators such as concatenation, repetition(zero or more and one or more) and union as well as intersection and complement should be applicable to both regular expressions and automata. The syntax for how to express these compositions must be designed and implemented.

2.4 Conversion

To facilitate composition using intersection and complement as well as the ability to choose the output format, one must be able to convert regular expressions to NFA's, convert NFA's to DFA's, minimise DFA's and convert automata back to regular expressions.

3 Design

3.1 Hybrid Syntax

To have a hybrid syntax of regular expressions and automata, one must have a way to express automata and regular expressions as well as a way to express how to compose them. To enable this the input is split into two: a grammar part where regular grammars, NFA, and DFA can be defined, and a regular expression part where one can refer to the non-terminals defined in the grammar part, and use them in standard regular expression compositional expressions.

3.1.1 Extended Regular Expressions

The syntax for extended regular expressions is as described below:

Symbol	Purpose	Usage
\	Escape next character	Anywhere except before alphanumerical characters
|	Union	Not in []
&	Intersection	Not in []
!	Complement	Not in []
*	Zero or more	Not in []
+	One or more	Not in []
?	Zero or one	Not in []
()	Grouping	Not in []
.	Any character in the alphabet	Not in []
[]	Start and end character class	Not in []
-	Range in character class	Only in []
^	Complement character class	At the beginning of []
# [a-zA-Z0-9]+	Reference non-terminal in grammar	Not in []

The following equivalencies are used to simplify the expression for future steps $s? = s|\epsilon$, $s^+ = ss^*$, $. = [\hat{}]$.

3.1.2 Grammar

To accommodate regular grammars, NFA, and DFA the syntax of the grammar follows that of a generalized NFA: non-terminals in the grammar represent states, and the production con-

sists of a single extended regular expression which represents the transition. Since extended regular expressions can contain non-terminals, non-terminal references can be used to create transitions to other states. Similarly, if a non-terminal has a production where the tail position is a terminal, then the state is accepting.

Note: Something about the actual format of the input? - grouped using brackets, arrows used to indicate productions and semicolons used to separate productions.
Automata -> '{' Grammar '}'
Grammar -> Nonterminal '->' ExtendedRegex ';' Grammar | ϵ
Nonterminal -> '#' [a-zA-Z0-9]+

3.2 Checking Well-formedness of Grammar

Since it is undecidable if a context-free grammar is regular, and the language represented by the grammar must be regular, it is necessary to place restrictions on the use of non-terminals in the grammar.

3.2.1 Restrictions

The tail position in a production is defined as the atom that occurs last. In the case of (r1|r2) the tail position is the tail position in both r1 and r2. Like in a regular grammar, circular references are only allowed in the tail position. This means that if a non-terminal occurs in a non-tail position in a production, then that non-terminal must not rely on the non-terminal whose productions it is in. Additionally, circular references are not allowed inside intersection, complement and "zero or more" expressions.

3.2.2 Algorithm

The algorithm for checking well-formedness of the grammar is divided into two parts: one part where the non-terminals are divided into layers where each layer contains non-terminals that are either mutually recursive, or only depend on non-terminals defined in previous layers. Additionally, this part checks whether the grammar is stratifiable - if it is not possible to divide all the non-terminals into layers, then the grammar is not stratifiable. The second part checks that all circular non-terminal references are in the tail position, and that no circular references occur inside intersection, complement and "zero or more".

3.2.2.1 Dividing into layers

Before trying to divide the non-terminals into layers, all productions for each non-terminal are collected such that each non-terminal only has a single production.

In every iteration of the algorithm it is attempted to build a new layer. It is done by iterating through all non-terminals and finding the non-terminals that exist in the non-terminal's production. Additionally, to account for mutual recursion, the dependencies of the dependencies are also found and added. If there are no dependencies, then the non-terminal can be added to the current layer immediately, otherwise check that all dependencies either exist in previous layers or are mutually recursive with the current non-terminal. If this is true for all dependencies, then the current non-terminal is added to the current layer.

If no non-terminals could be added to the layer, and the grammar is non-empty, then the grammar is not stratifiable, and the check for well-formedness fails. Otherwise the current non-terminal is removed from the grammar and the algorithm runs again with the updated grammar and layers. This process repeats until the grammar is empty or it is determined that the grammar is not well-formed.

3.2.2.2 Checking for legal use of non-terminals

Each layer contains only non-terminals that are either mutually recursive, or only depend on non-terminals defined in previous layers, with the latter never being part of a set of mutually recursive non-terminals. Thus, to ensure that recursion only occurs in the tail position, it is checked for each non-terminal in the grammar, that none of the non-terminals that exist in that non-terminal's layer occur in a non-tail position in that non-terminal's productions.

Since multiple tail non-terminals can exist, two initially empty sets of non-terminals are maintained: a set of non-tail non-terminals and a set of tail non-terminals. When analyzing a production, if there is a union, then tail positions are the tail positions in the two subexpressions, i.e the tail position of r1 and r2 in (r1|r2). If an expression consists of a sequence of terminals, then the tail position is simply the last terminal i.e. the tail position in (r1 r2) is the tail position of r2. In both cases, if any non-terminal from the same layer as the production's non-terminal is found, then the check for well-formedness fails. For complement, intersection and ”zero or more”, there is no need to distinguish between tail and non-tail positions, as any non-terminal occurence from the same layer as the production's non-terminal leads to the check for well-formedness failing.

3.3 Conversion

Conversion from input in the form of a generalized NFA and an extended regular expression to NFA is split into two parts. First, NFA templates are created for each layer, and every non-terminal in a layer is associated with a state in the template which is its starting state. This is done so when subsequent layers have references to non-terminals in a previous layer, or when the extended regular expression has a reference to a non-terminal, the NFA that corresponds to that non-terminal has already been computed, and can simply be copied and used. Afterwards, the extended regular expression can be converted to an NFA.

3.3.1 Regular expression to NFA

Conversion from regular expression to NFA uses the same algorithm as described in Mogensen, 2017 pages 9-11. NFA fragments are constructed from subexpressions and combined into bigger fragments. Every fragment has two incomplete transitions: one going into the fragment and one going out of the fragment, and these are used to combine different fragments. To model this, the intended end-state of a fragment is passed as an argument, and the regular expression is processed backwards, and each fragment returns its starting state, eliminating the need for the transition going into the fragment, as fragments can be combined using only the transition going out of the fragment. After combining all the fragments, the end-state that was initially passed to the algorithm can then be set as accepting, as this is the final state.

3.3.1.1 Standard regular expressions

3.3.1.2 Intersection

3.3.1.3 Complement

3.3.1.4 Non-terminals

3.3.2 Creating NFA templates

Before the algorithm can be applied, every union expression is removed and is replaced by two productions each consisting of one sub-expression each.

The algorithm iterates through all layers, and for each layer, a starting state is created for each of the non-terminals in that layer. Afterwards the productions for each non-terminal can be converted to NFAs.

When converting a non-terminal's productions to an NFA, each production is first converted individually to an NFA. If a production contains a non-terminal from the same layer, then that non-terminal is removed from the production, the starting state associated with that non-terminal is looked up, and the rest of the production is converted to an NFA using the algorithm described in 3.3.1 with the end-state of that NFA being the starting state associated with the removed non-terminal. Afterwards, for all the resulting NFAs, any accepting state that arises from the conversions is set to be rejecting, and an epsilon transition is added to the given end-state. Epsilon transitions are then added from the starting state associated with the non-terminal to the starting states of each of the NFAs resulting from converting the productions. Last, all sub-NFA's are combined

If a production contains a non-terminal from an earlier layer then, like in 3.3.1.4, the associated NFA and starting state are simply looked up and returned.

3.3.3 NFA to DFA

3.3.4 Minimisation of DFA's

3.3.5 Automata to regular expression

3.4 I/O

4 Implementation

5 Testing

6 User Guide

7 Conclusion

References

Mogensen, T. Æ. (2017). *Introduction to compiler design* (2nd). Springer.