UNIVERSITY OF COPENHAGEN

template210NFA template for first layerfigure.2 template310NFA template for second layerfigure.3

Bachelor's Project

Christian Bendix Fjordstrøm

Hybrid Syntax For Composition of Regular Languages

Date: June 4, 2024

Advisor: Andrzej Filinski

Abstract

Regular languages are fundamental to Computer Science and as such many students have to learn about the different representations of them. The objective of this project is to develop a tool that allows for composition of regular languages in the form of regular expressions, NFAs, DFA, and regular grammars using regular expression operators. Additionally, the tool provides the ability to convert the output to the various representation, making it useful for users desiring composition, conversion, or both.

Table of content

1 Introduction

2 Analysis

The project revolves around designing and implementing a tool that allows users to combine regular expressions, NFAs, DFAs and regular grammars by applying standard regular expression operations such as union, concatenation and Kleene star, as well as complement and intersection.

2.1 Syntax

The syntax should allow for arbitrarily nested compositional expressions consisting of the previous mentioned operations. The operations must be applicable to regular languages in any format, such that a user can, for example, determine if a regular expression and a DFA represent the same language by finding the complement of the regular expression and then finding the intersection of that language and the DFA. The syntax for how a user expresses these transformations must be designed and implemented.

2.2 I/O

The user should also be able to define which output format they want. 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 Conversion

To facilitate composition using intersection and complement as well as the ability to choose the output format, the program must be able to convert regular expressions to NFAs, convert NFAs to DFAs, minimise DFAs and convert automata back to regular expressions.

3 Design

This section covers the design choices that were made, and the algorithms that were used in the development of the tool.

3.1 Hybrid Syntax

To make it such that an expression can contain NFA, DFA, regular grammars and regular expressions the input is split in two. First is a grammar part with EBNF-like syntax that can be used to define automata and regular grammars. After is a regular expression part where one can refer to the non-terminals defined in the grammar part, and use the language they represent in standard regular expressions.

3.1.1 Extended Regular Expressions

The syntax of the extended regular expressions is shown in the grammar below:

$$\begin{array}{rl}
\langle \mathsf{ExtendedRegex} \rangle & \rightarrow \langle \mathsf{Seq} \rangle \\
 & \mid \langle \mathsf{Regex} \rangle \text{ “|” } \langle \mathsf{Regex} \rangle \\
 & \mid \langle \mathsf{Regex} \rangle \text{ “\&” } \langle \mathsf{Regex} \rangle \\
\langle \mathsf{Seq} \rangle & \rightarrow \epsilon \\
 & \mid \langle \mathsf{Rep} \rangle \; \langle \mathsf{Seq} \rangle \\
\langle \mathsf{Rep} \rangle & \rightarrow \langle \mathsf{Atom} \rangle \\
 & \mid \langle \mathsf{Atom} \rangle \text{ “*” } \\
 & \mid \langle \mathsf{Atom} \rangle \text{ “+” } \\
 & \mid \langle \mathsf{Atom} \rangle \text{ “?” } \\
 & \mid \text{ “!” } \langle \mathsf{Atom} \rangle \\
\langle \mathsf{Atom} \rangle & \rightarrow \langle \mathsf{Char} \rangle \\
 & \mid \text{ “(” } \langle \mathsf{ExtendedRegex} \rangle \text{ “)” } \\
 & \mid \langle \mathsf{Class} \rangle \\
 & \mid \langle \mathsf{Nonterminal} \rangle \\
\langle \mathsf{Char} \rangle & \rightarrow \text{[*+?()[]-.\textbackslash\#!]} \\
 & \mid \backslash [\hat{}a - zA - Z0 - 9] \\
\langle \mathsf{Class} \rangle & \rightarrow \text{“.”} \\
 & \mid \text{“[”} \langle \mathsf{ClassContent} \rangle \text{“]”} \\
 & \mid \text{“[”}\hat{}\langle \mathsf{ClassContent} \rangle \text{“]”} \\
\langle \mathsf{ClassContent} \rangle & \rightarrow \epsilon \\
 & \mid \langle \mathsf{ClassRange} \rangle \langle \mathsf{ClassContent} \rangle \\
\langle \mathsf{ClassRange} \rangle & \rightarrow \langle \mathsf{Char} \rangle \\
 & \mid \langle \mathsf{Char} \rangle \text{“} - \text{”} \langle \mathsf{Char} \rangle \\
\langle \mathsf{Nonterminal} \rangle & \rightarrow \text{“\#”}[a - zA - Z0 - 9]
\end{array}$$

The following equivalencies are used to simplify for future steps `s? = s`$|\epsilon$, `s`$^+$ `= ss`*, and `. = [^]`.

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 consists 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. The syntax can be seen below:

$$\begin{array}{rl}
\langle \mathsf{Automata} \rangle & \rightarrow \epsilon \\
 & \mid \text{“} \\
 & \\
\langle \mathsf{Grammar} \rangle & \rightarrow \epsilon \\
 & \mid \langle \mathsf{Nonterminal} \rangle \text{ “->” } \langle \mathsf{ExtendedRegex} \rangle \; \langle \mathsf{Grammar} \rangle \\
\langle \mathsf{Nonterminal} \rangle & \rightarrow \text{ “\#” [a-zA-Z0-9]}
\end{array}$$

3.2 Preprocessing the Grammar

The purpose of preprocessing is to first stratify the grammar, and then convert each of the resulting layers into NFA templates. This means that the grammar must be divided into layers where each layer contains non-terminals that are either mutually recursive, or only depend on non-terminals defined in previous layers. Instead of relying on the user to stratify the grammar, this is done automatically by the program. Since it is undecidable if a context-free grammar is regular, and the grammar must be regular, it is necessary to place restrictions on the use of non-terminals in the grammar.

After stratification, 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.

3.2.1 Restrictions on the use of non-terminals

The tail positions of a production is defined as the following:

- The tail position of an atom `a` is `a`
- The tail position of an intersection `(r1|r2)` are the tail positions in `r1` and `r2`
- The tail position of a concatenated expression `(r1 r2)` where `r2` is non-empty is the tail position in `r2`. If `r2` is empty, the tail position is the tail position in `r1`

Circular references are only allowed in the tail position, meaning that if there is a non-terminal `A` with production α, then circular references are allowed

- if the tail position of α is `A`
- if there is another non-terminal `B` with production $\mathtt{B} \rightarrow \beta\mathtt{A}$ then `A` can refer to `B` if the tail position of α is `B`

If `B` is not in the tail position of α, then `B` must not depend on `A`.

Any kind of circular references is prohibited in the following:

- Inside an intersection expression `(r1&r2)`
- Inside a complement expression `(!r)`
- Inside a Kleene star expression `(r*)`

3.2.2 Stratification Algorithm

The stratification algorithm is divided into two parts. In the first part the grammar is stratified, and in the second part it is checked that all circular non-terminal references are in the tail position, and that no circular references occur inside intersection, complement and Kleene star expressions.

3.2.2.1 Stratification

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 Stratification example

The following example shows the process of stratification:

```
{#a -> b #b;
#a -> d #d;
#b -> c #c;
#c -> a #a;
#d -> e #e;
#e -> 123;}
```

Figure 1: *Stratification example*

First all productions of each non-terminal are collected, so the `a` ends up with a single production `#a -> b #b | d #d`.

No layers have been defined. `e` is the only non-terminal that does not rely on other non-terminals, so it is added to the first layer and its production is removed. Next, since `d` is the only non-terminal that only depends on previous layers, and it is not mutually recursive with other non-terminals it is added to the second layer, and its production is removed. `a`, `b`, and `c` are mutually recursive, and `a` depends on `d` so `a`, `b`, and `c` are added to the final layer and their productions are removed. Since there are no more productions in the grammar, the stratification was successful.

3.2.2.3 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. The definition of what is in the tail position and what is the non-tail position follows the rules defined in 3.2.1.

To check for legal use: first, recursively find tail and non-tail non-terminals in the production. If the expression is

- Intersection `(r1|r2)`: if any non-terminal belonging to the same layer is in a non-tail position, then the grammar is not well-formed.
- Concatenation `(r1 r2)`: if any non-terminal belonging to the same layer is in a non-tail position, then the grammar is not well-formed.
- Intersection `(r1&r2)`: if any non-terminal belonging to the same layer exists in the expression, then the grammar is not well-formed.
- Complement `(!r)`: if any non-terminal belonging to the same layer exists in the expression, then the grammar is not well-formed.
- Kleene star `(r*)`: if any non-terminal belonging to the same layer exists in the expression, then the grammar is not well-formed.

3.2.2.4 Example of checking legal use of non-terminals

3.2.3 Creating NFA templates

The design of the algorithm that converts layers to NFA templates relies on the algorithm that converts regular expressions to NFAs. This is covered later in 3.3.1, but the essential part is that when converting a regular expression to an NFA, the end-state of the NFA is passed as an argument, and the starting state is returned along with a set of transitions.

3.2.3.1 Algorithm

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 arise from the conversions are set to be rejecting, and ϵ-transitions added to the given end-state. ϵ-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, the NFA associated with the layer that the non-terminal belongs to and starting state associated with that non-terminal are looked up, copied and inserted.

3.2.3.2 NFA template creation example

Applying the algorithm with '0' as the initial end-state to the example in 1 gives the following: The first layer is `e`. `e` is assigned '1' as its starting state. There are no unions to be removed, and no other non-terminals in its layer, so it is converted with '0' being the end-state. An ϵ-transition from `e`'s starting state to the starting state of the production is added, resulting in:

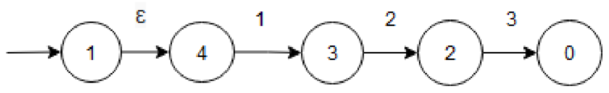


Figure 2: NFA template for first layer

The second layer is `d`. `d` is assigned '5' as its starting state. There are no unions to be removed. There is a non-terminal from a previous layer, so the template and starting state associated with `e` are looked up and copied. The rest of the production is converted, and an ϵ-transition from `d`'s starting state to the starting state of the production is added, resulting in:

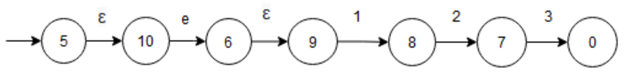


Figure 3: NFA template for second layer

The last layer is `a, b, c`. They are assigned ’11’, ’12’, ’13’ respectively as their starting states.

First, the productions of `a` are converted. `#a -> b #b` contains a reference to a non-terminal in the same layer so that non-terminal is removed and the production is converted to an NFA with the end-state being ’12’ - the starting state associated with `b`. This results in

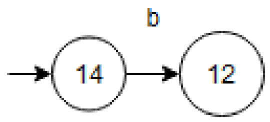


Figure 4: NFA template for #a -> b #b

`#a -> d #d` contains a reference to a non-terminal from a previous layer, so the template and starting state associated with `#d` are looked up and copied. The rest of the production is converted, resulting in:

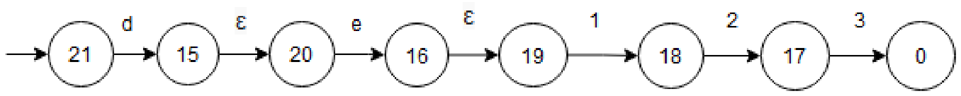


Figure 5: NFA template for #a -> d #d

ϵ-transitions are then added from the state associated with `a` - ’11’, to the starting state of each production giving the final NFA for `a`:

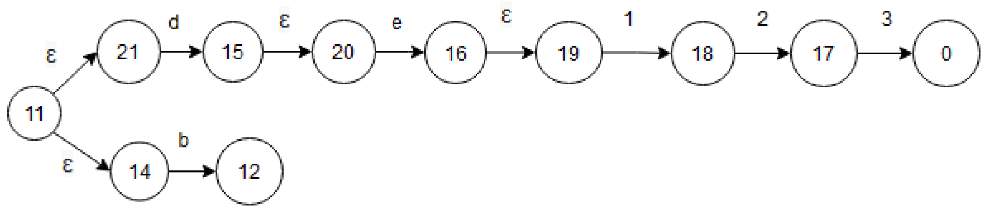


Figure 6: NFA template for #a -> b #b #a -> d .

For `#b -> c #b` and `#c -> a #a`, the same process is used as in `#a -> b #b`: the mutually recursive non-terminal is removed, and the regular expression is converted with the end-state being the starting state of `b` and `a` respectively. ϵ-transitions are then added from the starting states of `b` and `c`, resulting in the following NFAs:

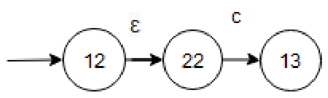


Figure 7: NFA template for #b -> c #c

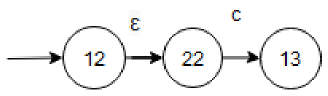


Figure 8: NFA template for #c -> a #a

Combining all the NFAs results in a final template:

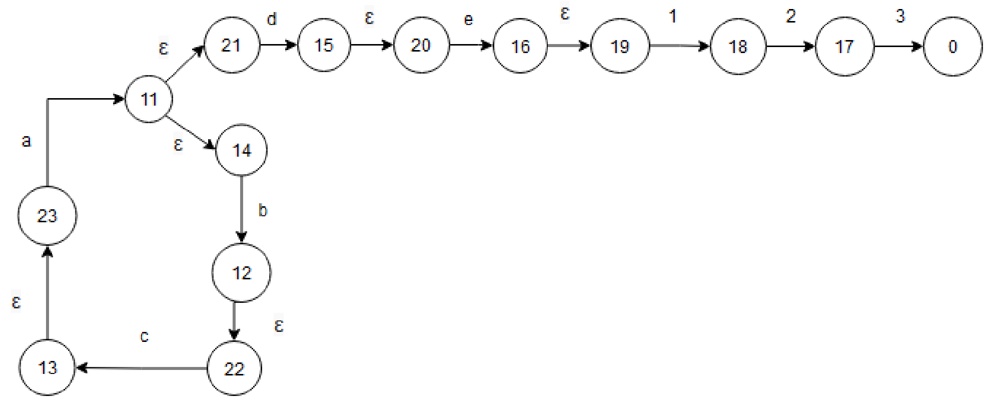


Figure 9: Final NFA template

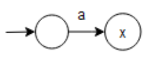
3.3 Conversion

3.3.1 Regular expression to NFA

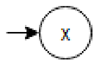
Conversion from regular expression to NFA uses the same algorithm as described in Mogensen, 2017 pages 9-11. NFA fragments are constructed from sub-expressions and combined into bigger fragments. Every fragment has two incomplete transitions: one going into the fragment and one going out of the fragment, and these are used to combine different fragments. To model this, the intended end-state of a fragment is passed as an argument, 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

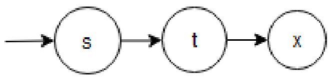
If ’x’ is passed as the end-state then the standard regular expression operators are converted to NFA fragments in the following way:



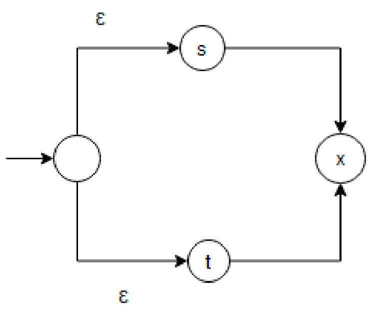
a



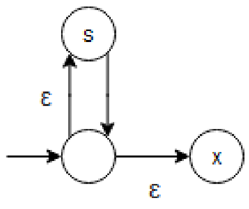
ϵ



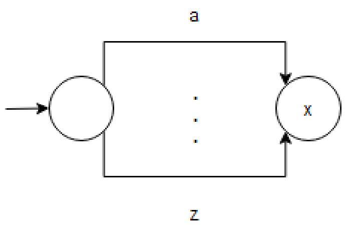
st



s|t



s*



[a-z]

3.3.1.2 Complement

The complement of an expression r is found by converting the expression to a DFA, making the DFA total by adding transitions to a dead state such that every state has a transition on every character in the alphabet and flipping accepting/rejecting flags on all states.

3.3.1.3 Intersection

The intersection of an expression s and an expression t is found using the product construction method. First s and t are converted to DFAs and made total using the same method as in 3.3.1.2.

A pair-state is created for every possible pair of states from s and t such that every pair consists of a state from s and a state from t. For every pair-state(s_x, t_x), for every symbol in the alphabet, s_x has a transition on that symbol to s_y, and t_x has a transition to t_y. For every symbol in the alphabet a transition is created from the pair-state (s_x, t_x) to (s_y, t_y) on that symbol. In a pair-state (s_x, t_x), if both s_x and t_x are accepting, then the pair-state is accepting, otherwise it is rejecting. The starting state is the pair consisting of the starting state in s and the starting state in t. The product-DFA is then converted to an NFA and returned.

3.3.1.4 Non-terminals

A non-terminal reference is converted by looking up the associated NFA-template and starting state and copying the NFA-template.

3.3.2 NFA to DFA

3.3.3 Minimisation of DFA's

3.3.4 Automata to regular expression

3.4 I/O

4 Implementation

5 Testing

The initial idea for testing was to use property-based testing by generating all possible inputs and testing the following properties: the intersection of an expression and its complement should be the empty language, the union of an expression and its complement should be zero or more of the union of all symbols in the alphabet. Due to time constraints, this was, however, not possible. Instead, a number of test cases are manually written, and the previously mentioned properties are tested. In addition to this, unit tests were created for each operation.

6 User Guide

6.1 Syntax

The syntax of the input can be seen in 3.1.1 and 3.1.2.

- `|` and `&` have the same precedence and are left-associative.
- Concatenation binds tighter than `|` and `&`.
- `*`, `+`, `?` and `!` bind tighter than concatenation

A table of which symbols have what purpose can be seen 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 []
ˆ	Complement character class	At the beginning of []
# [a-zA-Z0-9]$^+$	Reference non-terminal in grammar	Not in []

6.2 Grammar rules

In the grammar section of the input the following rules exist:

- Mutually recursive non-terminal uses are only allowed in the tail-position of a production
- Inside intersection, complement, and Kleene star expressions, no mutually recursive non-terminal uses are allowed
- The grammar must be stratifiable: it must be possible to divide the grammar into layer such that each layer contains non-terminals that are either mutually recursive, or only depend on non-terminals defined in previous layers.

6.3 Extended regular expression rules

If the extended regular expression contains complement of an expression(`!`), complement of a class(`[^]`) or "any symbol"(`.`), then an alphabet must be provided in the input.

6.4 Running the program from the command line

7 Conclusion

References

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