UNIVERSITY OF COPENHAGEN

Bachelor's Project

Christian Bendix Fjordstrøm

Hybrid Syntax For Composition of Regular Languages

Date: June 7, 2024

Advisor: Andrzej Filinski

Abstract

The objective of this project is to develop a tool that allows for composition of regular languages in the form of regular expressions, NFAs, DFAs, and regular grammars. The tool solves this by providing a hybrid syntax that allows users to express regular languages in the different representions, and compose them together in one unified syntax. 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 **5**

2 Analysis **6**

2.1 Syntax 6

2.2 I/O 6

2.3 Conversion 6

3 Design **6**

3.1 Hybrid Syntax 7

3.2 Preprocessing the Grammar 8

3.2.1 Stratifying the grammar 8

3.2.2 Stratification Algorithm 8

3.2.3 Creating NFA templates 10

3.3 Conversion 13

3.3.1 Regular expression to NFA 13

3.3.2 NFA to DFA 14

3.3.3 Minimisation of DFAs 15

3.3.4 Automata to regular expression 15

3.4 I/O 15

4 Implementation **16**

4.1 AbSyn 16

4.2 Lexing and Parsing 17

4.3 RegexToNFA 17

4.4 StratifyGrammar 17

4.5 NFAToDFA 17

4.6 MinimiseDFA 17
4.7 XFAToRegex 17
4.8 PrettyPrinter 17

5 Testing **17**

6 User Guide **18**

6.1 Syntax 18
6.2 Extended regular expression rules 18
6.3 Installation & Requirements 19
6.4 Running the program from the command line 19

7 Conclusion **19**

1 Introduction

The goal of this project is to develop a tool that enables composition of regular expressions, NFAs, DFAs, and regular grammars, and gives users the ability to transform input in one representation into output in another representation. To accomplish this, a hybrid syntax is developed that allows regular expressions, NFAs, DFAs and regular grammars to be composed together using regular expressions operators.

The following examples demonstrate how the tool can be used:

Standard regular expression can be defined and can be converted to a DFA and minimised:
`[a-zA-Z_][a-zA-Z_0-9]`*
can be converted to a minimised DFA giving:
`{`*#1* `->` `[a-zA-Z_]` *#2*`;`
#2 `->` `[a-zA-Z_0-9]` *#2* `|` `();}`
#1

A DFA can be converted back to a regular expression:
`{`#1 `->` `b` #2`;`
`{`#1 `->` `b` *#2*`;`
#1 `->` `d` *#4*`;`
#2 `->` `c` *#3*`;`
#3 `->` `a` *#1*`;`
#4 `->` `e` *#5*`;`
#5 `->` `123;}`
#1
can be converted to a regex giving:
`(bc(abc)`*`ade1|de1)23`

It can be checked if two languages are the same:
`{`*#1* `->` `a` *#2*`;`
#2 `->` `b` *#3*`;`
#3 `->` `c` *#4*`;`
#4 `->` `();`
`1&(abc)`
Which gives `[]`, meaning that they are the same.

One can even define a simple expression language:
`{`*#letter* -> `[a-z];`
#digit -> `[0-9];`
#word -> *#letter* `(`*#letter* `|` *#digit*`)`*`;`
#num -> *#digit*$^+$`;`
#kw -> `let | in;`
#var -> *#word* `& !`*#kw*`;`
#atom -> *#var*`;`
#atom -> *#num*`;`
#exp -> *#atom* *#rest*`;`

#rest -> `();`
#rest -> *#atom #rest*`;}`
#exp

2 Analysis

This section describes the considerations and requirements for the project.

2.1 Syntax

A way to accomplish the goal of the project could be to have separate programs that can transform between the different representations, and have another program that can compose expressions, however, this makes using the program complicated. The proposed solution in this project is to create a hybrid syntax where regular expressions, NFAs, DFAs and regular grammars can be defined, and every composition can be expressed in a single expression.

It is done by separating the input into two parts. First is a grammar part that can be used to define automata and regular grammars. The grammar is converted to NFA templates which are are way to associate every non-terminal with an NFA for the same language as the one the non-terminal represents.

After the grammar part is a regular expression part where one can refer to the non-terminals defined in the grammar part, and compose the languages they represent with standard regular expressions using regular expression operators.

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

The accomodate regular grammars, NFAs, and DFAs the syntax of the grammar part follows that of a generalized NFA: non-terminals in the grammar represent states, and the right-hand side consists of a single extended regular expression - standard regular expression extended with non-terminals, intersection, and complement - which represents the transition. Since extended regular expression can contain non-terminals, non-terminal references can be used to create transitions to other states. Similarly, if the right-hand side of a production has a terminal in the tail position, then that transition leads to an accepting state. The syntax of the hybrid syntax can be seen below. The production of some non-terminals are described using regular expressions.

ExtendedRegex	$\rightarrow$ *Seq*
	| *ExtendedRegex* '`|`' *ExtendedRegex*
	| *ExtendedRegex* '`&`' *ExtendedRegex*
Seq	$\rightarrow \epsilon$
	| *Rep Seq*
Rep	$\rightarrow$ *Atom* '`*`'
	| *Atom* '`+`'
	| *Atom* '`?`'
	| '`!`' *Atom*
Atom	$\rightarrow$ *Char*
	| '`'`' *ExtendedRegex* '`)`'
	| *Class*
	| *Nonterminal*
Char	$\rightarrow$ [$\widehat{\ }$`|*+?{}()-.#&!`]
	| '`\`' [`a-zA-Z0-9`]
Class	$\rightarrow$ '`.`'
	| '`[`' *ClassContent* '`]`'
	| '`[^`' *ClassContent* '`]`'
ClassContent	$\rightarrow$ *Char*
	| *Char* '`-`' *Char*
Automata	$\rightarrow \epsilon$
	| '`{`' *Grammar* '`}`'
Grammar	$\rightarrow$ *Nonterminal* '`->`' *ExtendedRegex* '`;`' *Grammar*
	| ϵ
Nonterminal	$\rightarrow$ '`#`' [`a-zA-Z0-9`]$^+$

'`|`' and '`&`' are both left-associative and have the same precedence. The following equivalencies are used to simplify for future steps s? = s|ϵ, $s^+ = ss^*$, and . = [$\widehat{\ }$].

3.2 Preprocessing the Grammar

3.2.1 Stratifying the grammar

Stratifying the grammar means dividing it into layers where each layer contains non-terminals that are either mutually recursive, or only depend on non-terminals that exist in previous layers. Instead of relying on the user to stratify the grammar, this is done automatically by the program. Additionally, 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. In the following sections, the tail position of the right-hand side of a production is a set of non-terminals and is defined as:

- The tail position of a non-terminal is itself.
- The tail position of a union (*r1*|*r2*) is the union of the tail positions in *r1* and *r2*.
- The tail position of a concatenated expression (*r1 r2*) where *r2* is not ϵ is the tail position in *r2*. If *r2* is ϵ, then the tail position is the tail position in *r1*.

The following example illustrates stratification:

```
{#1 -> b #2;
#1 -> d #4;
#2 -> c #3;
#3 -> a #1;
#4 -> e #5;
#5 -> 123;}
```

First, no layers have been defined. #1 is the only non-terminal that does not rely on other non-terminals, so it is added to the first layer. Next, since #4 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. #1, #2, and #3 are all either mutually recursive or only depend on non-terminals that exist in previous layers, so they are added to the final layer.

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 mutually recursive 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 Checking for legal use of non-terminals

For stratification to succeed it must be true for all layers that mutual recursion occurs only in the tail position. This means that if there is a production with non-terminal A and right-hand side α then mutually recursive references are allowed if

- the tail position of α is A
- there is another production $B \to \beta A$, then A can refer to B if the tail position of α is B

Additionally, no mutually recursive references are allowed inside

- Intersection expressions ($r1\&r2$)
- Complement expressions ($!r$)
- Kleene star expressions ($r*$)

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, the algorithm that checks for legal use of non-terminals maintains two initially empty sets of non-terminals: a set of non-tail non-terminals and a set of tail non-terminals. Tail and non-tail non-terminals are recursively found, and for each expression it is checked, that non-terminals found inside follow the previously outlined rules. If any of the checks fail, then stratification fails, and the program stops.

3.2.3 Creating NFA templates

An NFA template is created for each layer. Since stratification must have succeeded for this part to be reached, this part will always succeed.

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

As an example, algorithm is applied to the following grammar with '0' as the initial end-state.

```
{#1 -> b #2;
#1 -> d #4;
#2 -> c #3;
#3 -> a #1;
#4 -> e #5;
#5 -> 123;}
```

The first layer is *#5*. *#5* 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 *#5*'s starting state to the starting state of the production is added, resulting in:

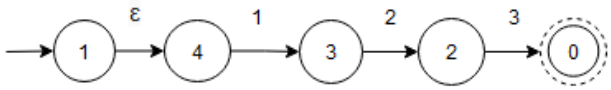


Figure 1: NFA template for first layer

The second layer is *#4*. *#4* 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 *#5* 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 2: NFA template for second layer

The last layer is *#1*,*#2*, and *#3*. They are assigned '11', '12', '13' respectively as their starting states.

First, the productions of *#1* are converted. *#1* `->` `b` *#2* 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 *#2*. This results in

Figure 3: NFA template for *#1* `->` `b` *#2*

#1 `->` `d` *#4* contains a reference to a non-terminal from a previous layer, so the template and starting state associated with *#4* are looked up and copied. The rest of the production is converted, resulting in:

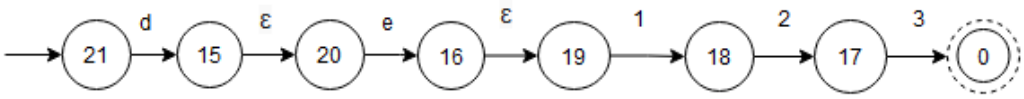


Figure 4: NFA template for *#1* `->` `d` *#4*

ϵ-transitions are then added from the state associated with *#1* - '11', to the starting state of each production giving the final NFA for *#1*:

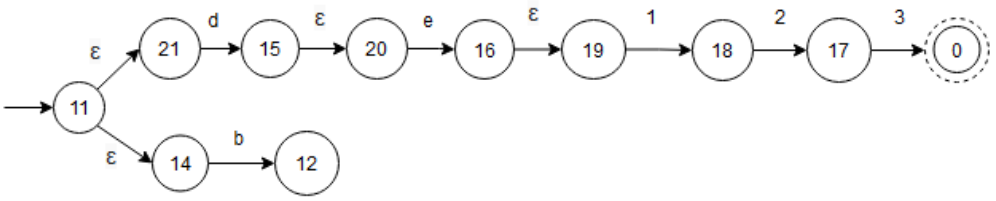


Figure 5: NFA template for *#1* `->` `b` *#2*`;` *#1* `->` `d` *#4*

For *#2* `->` `c` *#2* and *#3* `->` `a` *#1*, the same process is used as in *#1* `->` `b` *#2*: the mutually recursive non-terminal is removed, and the regular expression is converted with the end-state being the starting state of *#2* and *#1* respectively. ϵ-transitions are then added from the starting states of *#2* and *#3*, resulting in the following NFAs:

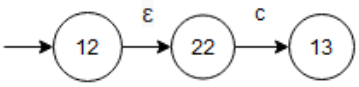


Figure 6: NFA template for *#2* `->` `c` *#3*

Figure 7: NFA template for *#3* `->` `a` *#1*

Combining all the NFAs results in a final template:

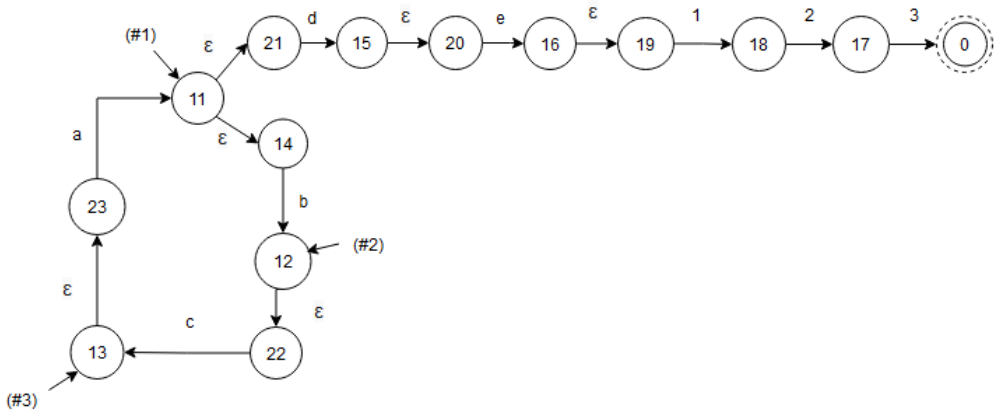


Figure 8: Final NFA template

3.3 Conversion

3.3.1 Regular expression to NFA

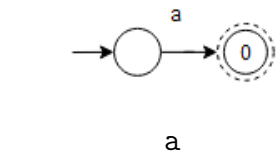
Conversion from regular expression to NFA uses the same algorithm as described in Mogensen, 2017 pages 9-11. NFA fragments are constructed from sub-expressions and combined into bigger fragments. Every fragment has two incomplete transitions: one going into the fragment and one going out of the fragment, 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 '0' is passed as the end-state then the standard regular expression operators are converted to NFA fragments in the following way:

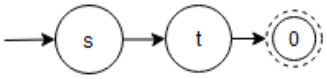
3.3.1.2 Complement

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.

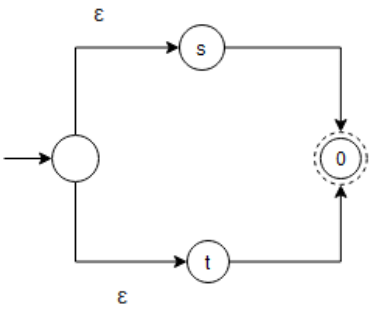


a

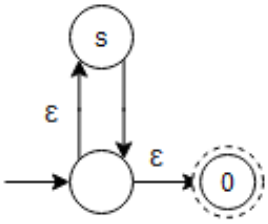
ϵ



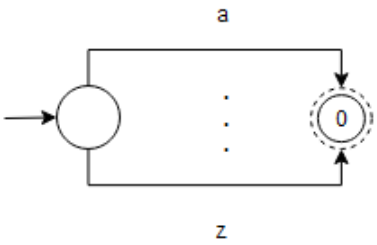
st



$s|t$



s^*



[a-z]

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

Conversion from NFA to DFA uses the subset construction algorithm as described in Mogensen, 2017 pages 16-19.

The algorithm maintains a work-list which associates a DFA state with its corresponding NFA states, and has a flag indicating if the DFA state is marked or not.

First, the epsilon closure of the starting state is computed and added to the work-list as unmarked. While there are unmarked states in the work-list, pick an unmarked state, and for each symbol

in the alphabet find the epsilon closure, referred to as s, of the states reachable by transitions on the current symbol. If there is no DFA state that is associated with s, then add an unmarked association between a new DFA state and s to the work-list. If s is empty, then there is no transition on that symbol, if s is non-empty, then there is a transition in the DFA to the DFA state associated with s. Last, all DFA states that map to accepting NFA states are accepting, the rest of the DFA states are rejecting.

3.3.3 Minimisation of DFAs

Minimisation of DFAs uses the algorithm described in Mogensen, 2017 pages 20-24. Dead states are handled by making it such that there are no undefined transitions. It is done by creating a new dead state and replacing all undefined transitions with transitions to this state.

A work-list is used which maps a minimal-DFA state to a set of DFA states, referred to as groups, and has a flag indicating if a minimal-DFA state is marked.

Initially the work-list has two unmarked entries: one group is the set of accepting states, and the other is the set of rejecting states. While there are unmarked and non-singleton groups pick one of these. The group is consistent if all transitions on the same symbol lead to the same group. If the group is consistent then it is marked and the you repeat. If the group is not consistent, then it is replaced by its maximal consistent subgroups - subgroups where all transitions on the same symbol lead to the same group.

3.3.4 Automata to regular expression

Automata are converted to regular expressions by using the idea of state elimination outlined in John E. Hopcroft and Ullman, 2001 pages 96-101. The idea of the algorithm is to remove states until there are only two states left, and for each removed state, introduce a regular expression that represents the transitions going in and out of that state.

If there is exactly one accepting state in the automata, then the transitions between states can be represented as a matrix, M where M_{ij} represents the transition going from state i to state j with M_{0j} and M_{i0} representing the incoming and outgoing transitions of the starting state, and M_{nj} and M_{jn} representing the incoming and outgoing transitions of the accepting state.

Thus, by converting an NFA, a new accepting state can be created, and all old accepting states can be given ϵ-transitions to the new accepting states. States can then be removed by iterating through the diagonal of the matrix, except for the starting and accepting state, and for every incoming transition(r_k) and for every outgoing transition(r_l) creating an expression $r1$ where, if the state has a transition to itself, then $r1 = r_k(r_{self})^* r_l$, and otherwise $r_1 = r_k r_l$. If there already is a transition r_2 in $M_k l$, then $M_{kl} = r_1|r_2$, otherwise $M_{kl} = r_1$.

3.4 I/O

The output is printed such that it has the same syntax as the input. This means that when printing automata, the transitions are printed in the same EBNF-like syntax as the transitions in the input, and the starting state is indicated using the regular expression part. Additionally, when

printing automata, if there are multiple transitions from one state to another, then all symbols are collected in a single character class, and accepting states are always printed using `()`. For example, the minimal DFA for `[a-c]` is printed as

```
{#1 -> [a-c] #2;
#2 -> ();}
#1
```

4 Implementation

The implementation of this project is approximately 1500 lines of F# code which is split into 12 modules: a lexer, a parser, a module containing the abstract syntax created by the parser, as well as other type definitions, a module to stratify the grammar, one that converts regular expressions to a NFA, one that converts an NFA to a DFA, one that converts a DFA to an NFA, one that minimises a DFA, one that converts either an NFA or a DFA to a regular expression, one that formats the output, one that tests if a given string is accepted by a given DFA, and a main program that calls other parts of the program, depending on what the user wants.

4.1 AbSyn

The abstract syntax of extended regular expressions is as follows:

```
1  type ClassContent = Set<char>
2
3  type Class =
4      ClassContent of ClassContent
5    | Complement of ClassContent
6
7  type ExtendedRegex =
8      Union of ExtendedRegex * ExtendedRegex
9    | Seq of ExtendedRegex * ExtendedRegex
10   | Class of Class
11   | ZeroOrMore of ExtendedRegex
12   | Nonterminal of string
13   | REComplement of ExtendedRegex
14   | Intersection of ExtendedRegex * ExtendedRegex
15   | Epsilon
```

This means that concatenated expressions are represented internally by combining binary **Seq** terms, and no argument ϵ terms. Additionally, single characters are represented as a special case of character classes - the case of a singleton class.

Using this definition of extended regular expressions, the grammar part of the input can be represented using the following type:

```
1  type Grammar = (string * ExtendedRegex) list
```

4.2 Lexing and Parsing

Lexing and parsing is done using FsLex and FsYacc respectively. When parsing, the optimisations explained in 3.1 are used to simplify the AST for future steps.

4.3 RegexToNFA

The type definition of an NFA is as follows:

```
1 type State = int
2 type Transition = char option * State
3 type NFAMap = Map<State, (Set<Transition> * bool)>
4 type Alphabet = Set<char>
5 type NFA = State * NFAMap * Alphabet
```

NFA states are represented internally as integers. Transitions are represented as char options a destination state, where **Some c** means that there is a transition on **c**, and **None** means that there is an ϵ-transition.

The module takes a regular expression as input, and returns the corresponding DFA.

4.4 StratifyGrammar

Layers are defined a list of string(non-terminal) lists:

```
1 type Layers = (string list) list
```

The module takes a grammar as input and, if stratification is possible, returns the resulting layers. If stratification is not possible, then an exception is raised and the program terminates.

4.5 NFAToDFA

4.6 MinimiseDFA

4.7 XFAToRegex

4.8 PrettyPrinter

5 Testing

The initial idea for testing was to include 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

- Converting a minimised DFA to a regular expression and then back to a minimised DFA should result in the same minimised DFA

Due to time constraints automatic, automatic generation of test data was 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 to test each operation.

6 User Guide

6.1 Syntax

The syntax of the input can be seen in 3.1.

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 []
^	Complement character class	At the beginning of []
# [a-zA-Z0-9]$^+$	Reference non-terminal in grammar	Not in []

6.2 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.

6.3 Installation & Requirements

6.4 Running the program from the command line

7 Conclusion

In this project, the primary objective was to design and implement a tool that enables composition of regular expressions, NFAs, DFAs, and regular grammars, as well allows users to choose the representation of the result.

A solution was proposed which uses a hybrid syntax consisting an EBNF-like grammar part, and an extended regular expression part which allows for intersection, complement, and non-terminal references.

If there was more time, then several operations could have been added such as Brzozowski derivatives, Left/right quotients, and reversal. When converting from automata to regular expresson, it could also have been attempted to create smaller regular expression by eliminating states in a suitable order. Last, automatic generation of test data for property-based testing could have been done.

In conclusion, this project succesfully designed and developed a tool that allows for composition of regular expressions, NFAs, DFAs, and regular grammars, and allows

References

John E. Hopcroft, R. M., & Ullman, J. D. (2001). *Introduction to automata theory, languages, and computation* (2nd). Addison-Wesley.

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