UNIVERSITY OF COPENHAGEN

Bachelor's Project

Christian Bendix Fjordstrøm

Hybrid Syntax For Composition of Regular Languages

Date: June 10, 2024

Advisor: Andrzej Filinski

Abstract

The objective of this project is to develop a tool that allows for composition and transformation of regular languages expressed 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 representations, 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 contents

1 Introduction

When learning how to implement a programming language, computer science students will often learn about lexical analysis. Since lexical tokens are typically defined using regular expressions, which are then converted to finite automata and used to divide the input string into tokens, students learn how to do these transformations. The project thus seeks to help students and teachers by providing a tool that, among others, makes it easy to answer if two regular languages are equal, find what they have in common, or simply convert from one representation to another. Use of the tool is illustrated in the following examples:

The regular expression `a*b|a` has the following minimal DFA:

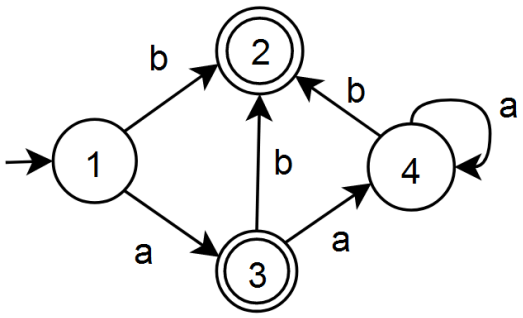

Figure 1: Minimal DFA for `a*b|a`

Running

```
$ dotnet run -mindfa a*b|a
```

will convert `a*b|a` to its minimal DFA, resulting in the following output:

```
{
#1 -> b #2 | a #3;
#2 -> ();
#3 -> b #2 | a #4 | ();
#4 -> b #2 | a #4;
}
#1
```

The output means that there is a state, #1, which has a b-transition to state #2 and an a-transition to state #3. `()` means that state #2 is accepting. #3 has a b-transition to #2, an a-transition to #4, and #3 is also accepting. #4 has a b-transition to #2, and an a-transition to #4. Last, the #1 after the closing bracket indicates, that #1 is the starting state of the DFA. As can be seen, this represents the same DFA as shown in figure 1.

DFAs can also be converted back into regular expressions. The DFA from the previous example can be converted back to a regular expression by running

```
$ dotnet run -regex {#1 -> b #2 | a #3; #2 -> (); #3 -> b #2 | a #4 | (); #4 -
> b #2 | a #4;}#1
```

which gives the following regular expression as output:

```
aaa*b|(a(b)?|b)
```

Looking at `a*b|a` and `aaa*b|(a(b)?|b)`, it is not obvious that these regular expressions represent the same language, however, by finding the symmetric difference, and checking that it is the empty language, one can check for equality of regular languages. This can be done by running

```
$ dotnet run -regex -alphabet ab {#1 -> aaa*b|(a(b)?|b); #2 -> a*b|a;}#1&!#2
```

which gives the following output:

```
[]
```

Since the hybrid syntax allows for regular expressions as the right-hand side of a production, defining a non-terminal can essentially be used as a form of let-binding. In this example, `&` means finding the intersection of two expressions, `!` means finding the complement of an expression, and the output, `[]`, represents the empty language. Since complement is used, it is necessary to provide the alphabet, which is done using the `-alphabet` flag.

A simple expression language consisting of letters and numbers can also be defined, and converted to a DFA:

```
$ dotnet run -mindfa -alphabet a-z0-9\+\-\*/ {#letter -> [a-z]; #digit -> [0-
9]; #word -> #letter (#letter | #digit)*; #num -> #digit+; #kw -> abs | sqrt |
log; #var -> #word&!#kw; #atom -> #var; #atom -> #num; #atom -> #kw #atom; #exp
-> #atom #rest; #rest -> (); #rest -> #binop #atom #rest; #binop -> \+ | \- |
\* | /;}#exp
```

which gives

```
{
#1 -> [0-9] #2 | a #3 | l #4 | [b-km-rt-z] #5 | s #6;
#2 -> [\*\+\-/] #1 | [0-9] #2 | ();
#3 -> [\*\+\-/] #1 | [0-9ac-z] #5 | b #7 | ();
#4 -> [\*\+\-/] #1 | [0-9a-np-z] #5 | o #8 | ();
#5 -> [\*\+\-/] #1 | [0-9a-z] #5 | ();
#6 -> [\*\+\-/] #1 | [0-9a-pr-z] #5 | q #9 | ();
#7 -> [\*\+\-/] #1 | [0-9a-rt-z] #5 | s #10 | ();
#8 -> [\*\+\-/] #1 | [0-9a-fh-z] #5 | g #10 | ();
#9 -> [\*\+\-/] #1 | [0-9a-qs-z] #5 | r #11 | ();
```

```
#10 -> [0-9a-z] #5;
#11 -> [\*\+\-/] #1 | [0-9a-su-z] #5 | t #10 | ();
}
#1
```

While this example cannot be used to produce an abstract syntax tree of the expression language, it could be used to check that an expression is well-formed by running
`$ dotnet run -run <input string> <regular language>`
In the examples it can be seen, that regular expression operators such as $^+$, *, and `&` can be applied to non-terminals that can represent automata, which is how the different representations of regular languages can be composed together, repeated etc.

1.1 Related Work

A similar hybrid syntax as the one presented in this project has been designed for the Kleenex language, described in Grathwohl et al., 2016. Additionally, tools such as JFLAP(Rodger, n.d.) and FSM2Regex(Zuzak, n.d.) exist that can be used to convert between different representations of regular languages.

1.2 Overview

In Section 2 the considerations and requirements for the project are described. Afterwards, the syntax of the hybrid syntax is introduced, and the algorithms used to compose and transform regular expressions, NFAs, DFAs, and regular grammars are explained (Section 3). Implementation details (Section 4) and how the tool was tested (Section 5) are then covered, followd by Section 6 which is a guide for how to use the tool. Section 7 concludes the report, and discusses possible future work.

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 using standard 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

3.1 Hybrid Syntax

To 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 of a production represents a transition, and consists of a single regular expression extended with non-terminal references, intersection, and complement. Since regular expressions 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.

Start	→ *Grammar Regex*
Regex	→ *Seq*
	| *Regex* '`|`' *Regex*
	| *Regex* '`&`' *Regex*
Seq	→ ϵ
	| *Rep Seq*
Rep	→ *Atom* '`*`'
	| *Atom* '`+`'
	| *Atom* '`?`'
	| '`!`' *Atom*
Atom	→ *Char*
	| '`(`' *Regex* '`)`'
	| *Class*
	| *Nonterminal*
Char	→ [`^|*+?{}()-.\#&!`]
	| '`\`' [`^a-zA-Z0-9`]
Class	→ '`.`'
	| '`[`' *ClassContent* '`]`'
	| '`[^`' *ClassContent* '`]`'
ClassContent	→ *Char*
	| *Char* '`-`' *Char*
Grammar	→ ϵ
	| '`{`' *Productions* '`}`'
Productions	→ *Nonterminal* '`->`' *Regex* '`;`' *Productions*
	| ϵ
Nonterminal	→ '`#`' *Alphanums*
Alphanums	→ [`a-zA-Z0-9`]
	| [`a-zA-Z0-9`] *Alphanums*

The following characters are ignored in the input: white space, tab, carriage return, newline,

and form feed. '|' and '&' are both left-associative and '&' binds tighter than '|'. The following equivalencies are used by the parser to desugar the input for future steps s? = s|ϵ, s^+ = ss^*, and . = [^]. Desugaring s^+ to ss^* can cause duplication of s, as an optimized method for converting s^+ to an NFA cannot be used, however, because this simplifies other steps, the desugaring is still performed.

3.2 Preprocessing the Grammar

3.2.1 Definition of a stratified grammar

The term stratified grammar will be used to describe a grammar that has been divided into layers such that each layer only contains non-terminals that are either mutually recursive, or only depend on non-terminals that exist in previous layers.

In the following sections, the tail positions of a regular expression 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 tail position of any other expression is the empty set.

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. Therefore, 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 $B \rightarrow \beta A$, 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 ($r*$)

3.2.2 Stratifying the grammar

Instead of relying on the user to stratify the grammar, this is done automatically by the program.

The following example illustrates stratification:
```
{#1 -> b #2 | d #4;
#2 -> c #3;
#3 -> a #1;
#4 -> #5 #5;
#5 -> 123;}
```

First, no layers have been defined. #5 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. The resulting stratified grammar is {#5},{#4}, {#1, #2, #3}.

3.2.3 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 mutually recursive references occur inside intersection, complement and Kleene star expressions.

3.2.3.1 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 stratification 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 stratifiable.

3.2.3.2 Checking for legal use of non-terminals

After diving the non-terminals into layers, it is know that 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 rules outlined in 3.2.1. If any of the checks fail, then stratification fails, and the program stops.

3.2.4 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.

3.2.4.1 NFA template creation algorithm

The algorithm takes a stratified grammar as input, and returns a list of 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. For example

```
{#1 -> a | b;}
```

would become

```
{#1 -> a;
#1 -> b;}
```

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 states that arise from the conversions are set to be rejecting, and ϵ-transitions are added from the old accepting state to the given end-state. If the production does not contain a non-terminal from the same layer, then it is simply converted to an NFA using the algorithm described in 3.3.1. 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.

3.2.4.2 NFA template creation example

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

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

In the previous example it was shown that stratification of the grammar results in: {#5},{#4}, {#1, #2, #3}.

The first layer contains only *#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:

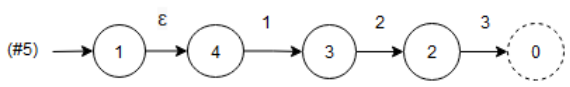

Figure 2: NFA template for first layer

The second layer contains *#4*. *#4* is assigned '5' as its starting state. There are no unions

to be removed. There are non-terminals from previous layers, so the template and starting state associated with *#5* are looked up and copied, when *#5* is encountered the second time, the starting state and NFA-template are again looked up and copied. Last, an ϵ-transition from *#d*'s starting state to the starting state of the production is added, resulting in:

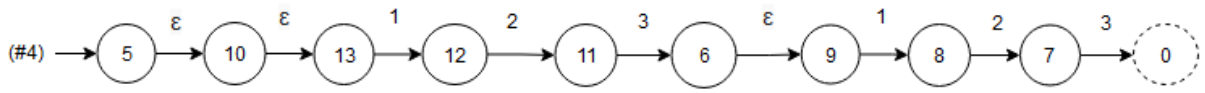

Figure 3: NFA template for second layer

The last layer contains *#1*,*#2*, and *#3*. They are assigned '14', '15', '16' respectively as their starting states.

First, the production of *#1* is converted. The right-hand side contains a union, so it is split into two productions: *#1* `->` `b` *#2* and *#1* `->` `d` *#4*.

#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 '15' - the starting state associated with *#2*. This results in

Figure 4: 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:

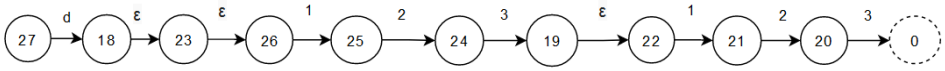

Figure 5: NFA template for *#1* `->` `d` *#4*

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*:

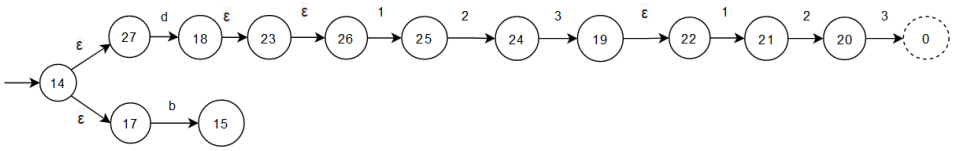

Figure 6: NFA template for *#1* `->` `b` *#2*`;` *#1* `->` `d` *#4*

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:

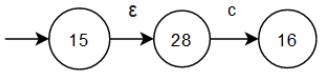

Figure 7: NFA template for *#2* `->` `c` *#3*

Figure 8: NFA template for *#3* `->` `a` *#1*

Combining all the NFAs results in a final template:

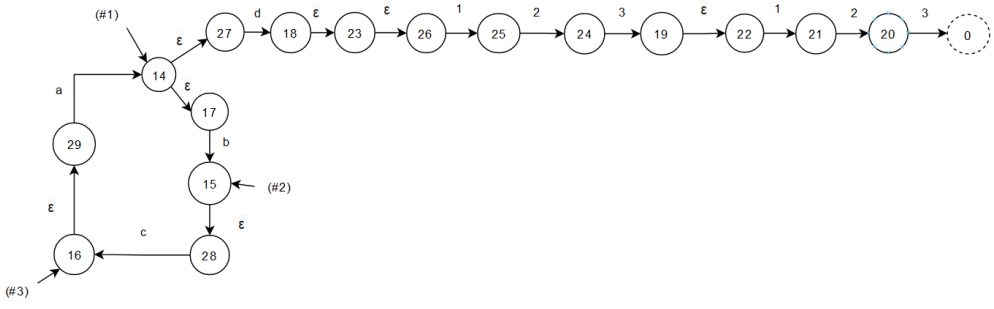

Figure 9: Final NFA template

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 Section 1.3. 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 can be set as accepting, as this is the final state.

The following example illustrates how `st`* would be converted using this algorithm:

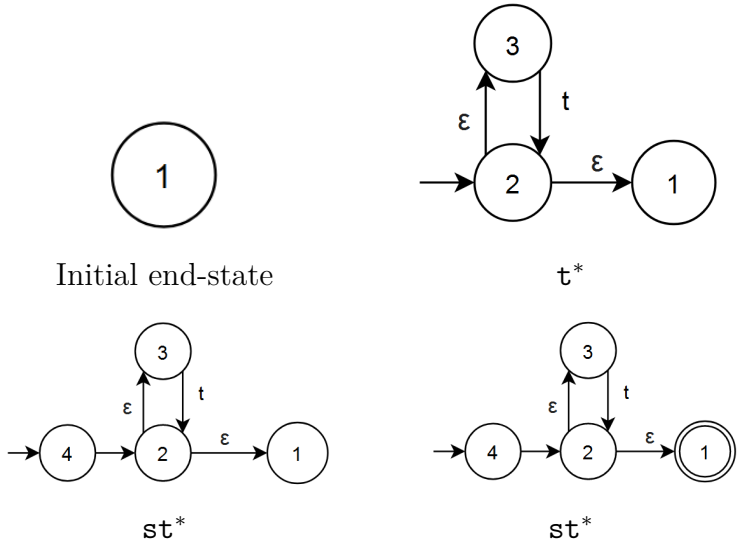

Figure 10: Conversion of `st`* to an NFA

First, an initial end-state is created, then `t`* is converted with the initial end-state as its end-state, then `s` is converted with the starting state of `t`* as its end-state, and then the initial end-state is set as accepting.

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:

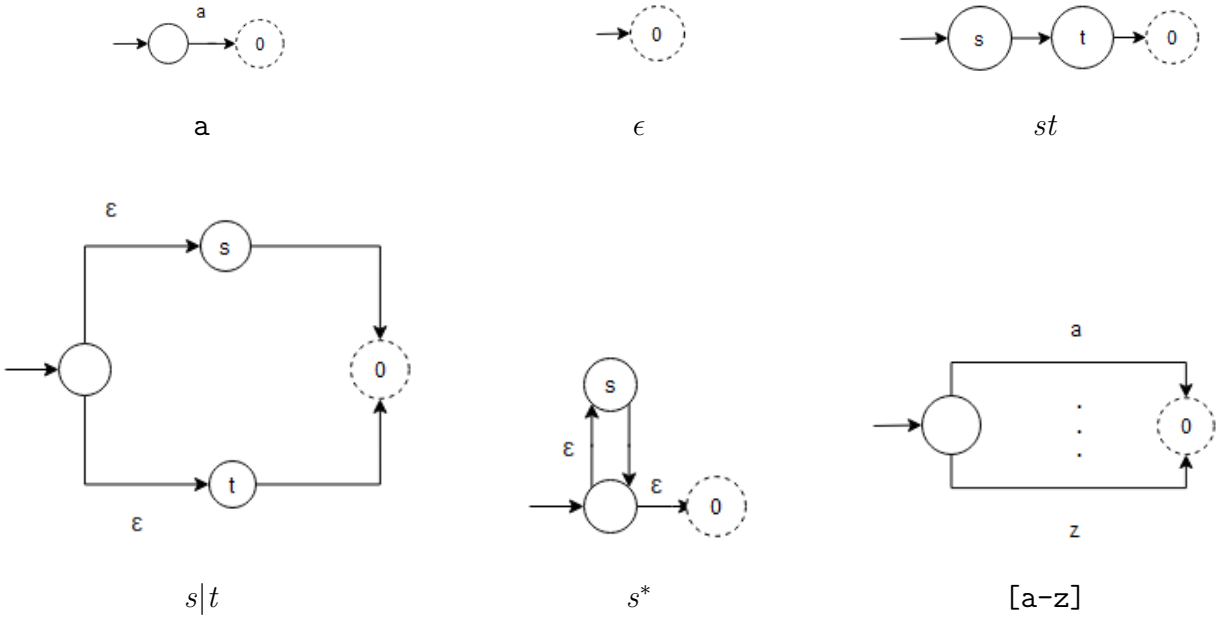

Figure 11: Graphical representation of how standard regular expression operators are converted to NFA fragments

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

Conversion from NFA to DFA uses the subset construction algorithm as described in Mogensen, 2017 Section 1.5.

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 Section 1.7. 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 Section 3.2.2. The idea of the algorithm is to convert the automata to a generalized NFA, an NFA with regular expressions as symbols on transitions, and 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_{in} 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 `e|c|b|a` is printed as

```
{#1 -> [a-ce] #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. States are associated with transitions by using a map that maps NFA states to a set of transitions, and states are also associated with a boolean value indicating if the state is accepting or not. An NFA can then be defined as a starting state, a mapping of states to transitions a flag indicating if the state is accepting or not, and the alphabet for that NFA.

Since conversion to DFA is needed for intersection and complement, the type of a DFA is also relevant in this module. Because DFAs need to be used for product construction, it is useful if a DFA state can be a pair of NFA states. Therefore, a DFA is parameterised on the type of its state. Additionally, to support the ability to check if a DFA accepts a given string, a DFA state's transitions are a map from chars to DFA states. The type definition is as follows:

```
1 type DFA<'State when 'State : comparison> = 'State * Map<'State, (Map<char, 'State> *
      bool)> * Set<char>
```

This module takes a regular expression as input, and returns the corresponding NFA.

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

The work-list is implemented as a map from states to sets of states and a bool that indicates if the state is marked or not.

This module takes an NFA as input and returns a DFA.

4.6 XFAToRegex

To represent the generalized NFA as a matrix, the data type should model a matrix with optional regular expression entries. The following type is used for this purpose:

```
1 type GNFA = ExtendedRegex option[,]
```

The module takes either an NFA or a DFA as input, and returns the corresponding regular expression.

5 Testing

The initial idea for testing was to include property-based testing by generating a variety inputs and testing e.g. 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 the language of all words over 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 feasable. Instead, a number of test cases were manually written, and the previously mentioned properties were 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

Running the tool requires .NET 7.0 or later. The source code is available on GitHub at https://github.com/CFjordstrom/Bachelor, and can be installed and built by running the following commands:

```
$ git clone https://github.com/CFjordstrom/Bachelor.git
$ make
```

6.4 Running the program from the command line

After building, unit tests can be run using
`$ ./run_unit_tests.sh`

and property based tests can be run using
`$ ./run_property_tests.sh`

A custom input can be run by navigating to the `src` folder and running
`$ dotnet run <output option> <filename or regular language>`

Possible output options are: `-regex`, `-mindfa`, `-dfa`, and `-nfa`. An alphabet can be provided after the output option using `-alphabet <alphabet>`.

It can also be checked if a string is accepted by the regular language by running
`$ dotnet run -run <input string> <filename or regular language>`

7 Conclusion

In this project, the primary objective was to design and implement a tool that enables composition and conversion of regular expressions, NFAs, DFAs, and regular grammars.

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. Every operation was tested and shown to work correctly, which gives confidence that the tool works correctly, and that it can be useful for everyone that wants to combine and convert regular languages.

Possible future work includes adding more regular language operations such as

- left/right quotients - removing either a specified prefix or suffix from a regular language
- reversal of a regular language

Other additions could also have been made such as

- 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
- automatic generation of test data for property-based testing

References

Grathwohl, B. B., Henglein, F., Rasmussen, U. T., Søholm, K. A., & Tørholm, S. P. (2016). Kleenex: Compiling nondeterministic transducers to deterministic streaming transducers. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

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.

Rodger, S. H. (n.d.). Jflap. https://www.jflap.org/

Zuzak, I. (n.d.). Fsm2regex. https://ivanzuzak.info/noam/webapps/fsm2regex/