DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exercises on regular expressions in Automata Tutor

Emanuel Ramneantu

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exercises on regular expressions in Automata Tutor

Aufgaben zu regulären Ausdrücken in Automata Tutor

Author:	Emanuel Ramneantu
Supervisor:	Prof. Jan Křetínský
Advisor:	M.Sc. Maximilian Weininger
Submission Date:	April 15, 2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, April 15, 2019 Emanuel Ramneantu

Abstract

This thesis presents the additions made to the Automata Tutor learning platform for supporting regular expressions. This includes alterations of one preexisting exercise type, Description to Regular Expression (DescriptionToRegex), as well as three completely new exercises, namely finding words recognized by a regular expression (WordsInRegex), converting a regular expression to a Nondeterministic Finite Automaton (RegexToNfa) and an exercise on equivalency classes (EquivalenceClasses). For each of the problem types three views have been created, two instructor views for creating and editing the problem statement and one student view for solving the problem. A graphical user interface has been developed for the RegexToNfa exercise to allow the the input of a richer type of automata, which we call Block Automata. Additionally, each problem type can give feedback and a grade to the students after the submission of their solution. This thesis describes in detail each part of the implementation and gives insight into the decisions that were taken along the way.

Contents

Chapter 1

Introduction

1.1 Motivation

Regular expressions are used for the definition of a language in a variety of fields. This makes them perfect for specifying the structure of common pieces of text. There are many examples of this. In the field of computer science regular expressions help define tokes used by compilers in order to assess whether the text follows the syntax of the programming language. In a less technical setting a regular expression can be used to define a valid price (e.g. $17.85) to make sure a text field is filled out appropriately. This is done by defining a format, in our case a dollar sign followed by an arbitrary number of digits with no leading zeros and optionally a point followed by two more digits.

The goal of this thesis is to provide an interactive and attractive way to learn about regular expressions and their strong connection to regular languages though different exercise types. In the context of Automata Tutor this is one more piece of the puzzle towards creating an automatic evaluation tool for formal languages.

The exercise types presented in this thesis allow for a deeper understanding of regular expressions and their connection to regular languages and finite automata. The difficulty varies from one exercise type to another and therefore allows for the study of regular expressions at different levels. For instance, the problem type WordsInRegex is easier than DescriptionToRegex since the former only requires giving words that are recognized by the regular expression as opposed to specifying a regular expression that recognizes a language given informally by a description. The problem type RegexToNfa makes the connection to finite automata which have the same expressive power but a completely different structure. That makes this problem type a good stepping stone into the more theoretical side of formal languages where there is often the need to prove equivalency. Finally, the problem type EquivalenceClasses is designed to inspect whether two words are equivalent with respect to a given regular expression. This is arguably the most difficult problem and requires some additional knowledge about regular languages.

1.2 Previous Work

Automata Tutor [DWWA15] is a teaching tool which provides numerous exercises to the fundamental concepts of theoretical computer science [Sch92, HMU06]. This thesis is the latest in a series of expansions of Automata Tutor, which have brought many new problem types to the platform. At the present moment Automata Tutor covers all the major grammar types in the Chomsky hierarchy, except for Type-1. Type-0 grammars are represented by the WHILE-Program to Turing Machine (TM) exercise type [Bac18, Ajd18]. Type-2 grammars are exemplified with the English to Pushdown Automaton (PDA) and Words in PDA exercise type [Mai18], as well as the Grammar to Chomsky normal form (CNF) and CYK Algorithm exercise types [Hel17]. Most problem types are for Type-3 grammars, including but not limited to: NFA to Deterministic Finite Automaton (DFA) and product construction [Wag17], Description to DFA, Description to Grammar, Pumping Lemma [Gor18].

1.3 Overview

This thesis presents four different exercise types implemented in the teaching tool Automata Tutor. Chapter 2 includes the theoretical background for understanding the methods used in creating these exercise types. Chapter 3 presents all the exercise types in detail, with implementation aspects relevant to the programmer as well as indications for the end user on how to interact with the software. Chapter 4 introduces the graphical user interface (GUI) developed for Block Automata. Chapter 5 contains the results of performance benchmarks on the RegexToNfa problem type in an attempt to find if it behaves as expected under intense load. In the 6th and final chapter conclusions are drawn and an outlook towards the future development of Automata Tutor is presented.

Chapter 2

Preliminaries

2.1 Theoretical background

This chapter presents some foundational knowledge about regular languages in order to understand the notation and methods used in the rest of the thesis. Unless stated otherwise, all of the following definitions, theorems and corollaries are due to [Sch92]. For further detail, consult [Sch92, HMU06].

An *alphabet* is a finite set. The elements of the set are called symbols. A *word* is a finite sequence of symbols of an alphabet. ϵ denotes the *empty word*, which contains no symbols. ϵ is a valid word for any alphabet. The *length* of a word is given by the number of positions for symbols in that word. ϵ has length zero. If Σ is an alphabet, Σ^k represents the set containing all the words of length k over Σ [HMU06, p. 28–30].

Definition 2.1 (Kleene Star [HMU06])**.** Let Σ be an alphabet. Then:

$$\Sigma^* := \bigcup_{k \in \mathbb{N}_0} \Sigma^k$$

Intuitively Σ^* is the set of all words over Σ, including ϵ. We let $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$, i.e. the set of words of length greater than zero.

Example 2.2. Let $\Sigma = \{a, b\}$. Then:

- $\Sigma^0 = \{\epsilon\}$
- $\Sigma^2 = \{aa, ab, ba, bb\}$
- $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
- $\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, \dots\}$

Definition 2.3 (Formal Language)**.** Let Σ be an alphabet. A formal language (over Σ) is any subset of Σ^*.

Definition 2.4 (Grammar)**.** A Grammar is a tuple $G = (V, \Sigma, P, S)$ where:

- V is a finite set, the set of variables.
- Σ is a finite set, the set of terminal symbols. It must hold that $V \cup \Sigma = \emptyset$.
- P is a finite subset of $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$, the production rules.
- $S \in V$ is the starting variable.

Let $u, v \in (V \cup \Sigma)^*$. The relation $u \Rightarrow_G v$ is defined if:

$$\begin{array}{lcl} u & = & xyz \\ v & = & xy'z \text{ where } x, z \in (V \cup \Sigma)^* \\ y & \rightarrow & y' \text{ is a production in } P \end{array}$$

Then the language defined by G is: $L(G) = \{w \in \Sigma^* | S \Rightarrow_G^* w\}$ where $\Rightarrow_G^*$ is the reflexive and transitive Hull of $\Rightarrow_G$.

Definition 2.5 (Regular Language)**.** A Grammar is of Type-3 or regular if the following holds:

- if $w_1 \rightarrow w_2$ is in P, then $|w_1| \leq |w_2|$.
- $w_1 \in V$, i.e. w_1 is a single variable.
- $w_2 \in \Sigma \cup \Sigma V$, i.e. the right side of the productions is a terminal symbol or a terminal symbol followed by a variable.

Let G be a regular grammar. Then $L(G)$ is a regular language.

Constraints imposed on the productions of a grammar categorize languages into four types, which form the Chomsky hierarchy. Aside from regular grammars (Type-3), there are Type-0 grammars, equivalent to Turing Machines, context-sensitive grammars (Type-1) and context-free grammars (Type-2). In the following, we are only concerned with regular languages. Apart from grammars, there are multiple ways to represent regular languages. We consider regular expressions and four types of automata in this thesis, all of which are introduced now.

Example 2.6. Suppose $V = \{S, A, B, C\}$, $\Sigma = \{a, b, c\}$ and P contains the following productions:

$$\begin{array}{lcl} S & \rightarrow & aA \\ A & \rightarrow & aA \mid bB \mid aC \\ B & \rightarrow & bB \mid b \\ C & \rightarrow & c \end{array}$$

Then $G = (V, \Sigma, P, S)$ is a regular grammar.

Definition 2.7 (Deterministic Finite Automaton)**.** A DFA M is defined by the tuple $M = (Z, \Sigma, \delta, z_0, E)$ where:

- Z is the finite set of states.
- Σ is the alphabet, $Z \cap \Sigma = \emptyset$.
- $z_0 \in Z$ is the initial state.
- $E \subseteq Z$ is the set of final states.
- $\delta : Z \times \Sigma \rightarrow Z$ is the transition function.

$\mathcal{D}$ is the set of all DFAs.

Definition 2.8 (Language of a DFA)**.** Given a DFA $M = (Z, \Sigma, \delta, z_0, E)$, let $\hat{\delta} : Z \times \Sigma^* \rightarrow Z$ be defined recursively:

$$\begin{array}{rcl} \hat{\delta}(z, \epsilon) & = & z \\ \hat{\delta}(z, ax) & = & \hat{\delta}(\delta(z, a), x) \end{array}$$

where $z \in Z, x \in \Sigma^*$ and $a \in \Sigma$.
The language L of M is the given by the following set. We also say that M recognizes $L(M)$.

$$L(M) = \{w \in \Sigma^* | \hat{\delta}(z_0, w) \in E\}$$

Figure 2.1 illustrates a DFA. DFAs can be represented as graphs, with the nodes of the graph symbolizing the states and the edges symbolizing the transitions. Notice that all the transitions are defined. Final states are marked by a black border and the initial state is marked by an unlabeled short arrow. The function $\hat{\delta}$ can be imagined as a path through the graph, reading the symbols of the word in order and traversing a link with the corresponding label for each of them. We say an automaton (or a grammar, regular expression) M accepts a word w if $w \in L(M)$. Otherwise, the automaton rejects the word. Two representations of formal languages are called equivalent if they recognize the same language.

Example 2.9. The DFA $M = (Z, \Sigma, \delta, z_0, E)$ illustrated in Figure 2.1 accepts the word *aac*, because the path $1 \to 2 \to 3 \to 5$ leads to an accepting state. More formally, $\hat{\delta}(z_0, aac) \cap E = \{5\} \neq \emptyset$. Similarly, it rejects the word *aaaa*, because the path leads to the state 4.

It is worth noting that $L(M) = L(G)$, where G is the grammar from Example 2.6.

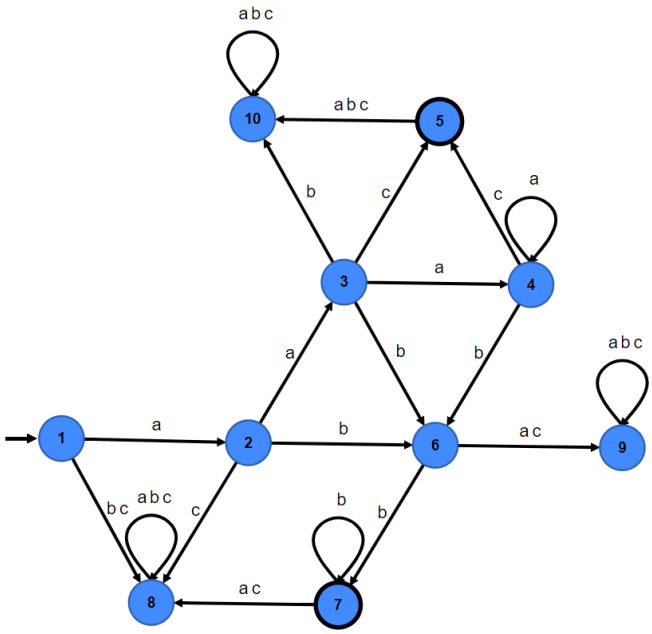

Figure 2.1: A DFA

Theorem 2.10. Every language that is recognizable by a DFA is regular (of Type-3).

Proof. see [Sch92, p. 30] □

Definition 2.11 (Nondeterministic Finite Automaton)**.** An NFA M is defined by the tuple $M = (Z, \Sigma, \delta, S, E)$ where:

- $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ is the transition function.
- $S \subseteq Z$ is the set of initial states.
- Z, Σ, E have the same definition as for DFA (see Definition 2.7).

An NFA allows more than one transition per symbol from every state. As such, the range of δ is a set of sets. It is also not necessary to define every transition. This often simplifies the automaton. If *all* paths for a word arrive at some point in a state with no outgoing transition for the current symbol or lead to a non-final state, the word is rejected by the automaton.

Example 2.12. The NFA $M = (Z, \Sigma, \delta, S, E)$ illustrated in Figure 2.2 accepts the word *aabb* because *there exists a path* starting from state 1 and leading to state 6 while following the links for the letters of *aabb*. Note

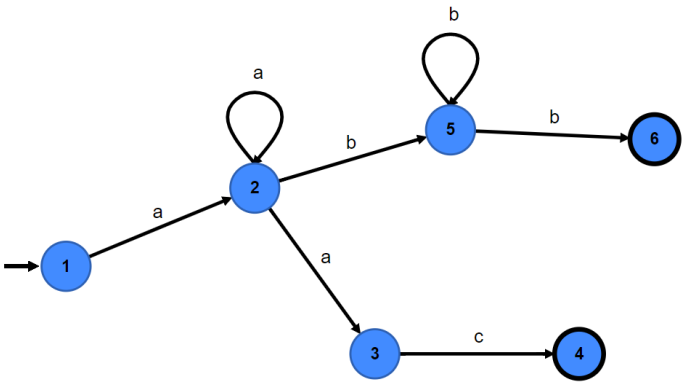

Figure 2.2: An NFA

that there also exists a path, $1 \to 2 \to 2 \to 5 \to 5$, that does not lead to an accepting state. There also exists a path, $1 \to 2 \to 3 \to ?$, which gets stuck in state 3.

It holds that $L(M) = L(G)$, where G is the grammar from Example 2.6. This also make it equivalent to the DFA from Figure 2.1.

Definition 2.13 (Language of an NFA)**.** The function δ can once again be generalized to $\hat{\delta} : \mathcal{P}(Z) \times \Sigma^* \to \mathcal{P}(Z)$ as follows:

$$\begin{aligned} \hat{\delta}(Z', \epsilon) &= Z' \text{ for all } Z' \subseteq Z \\ \hat{\delta}(Z', ax) &= \bigcup_{z \in Z'} \hat{\delta}(\delta(z, a), x) \text{ where } a \in \Sigma \text{ and } x \in \Sigma^* \end{aligned}$$

Then the language L of M is given by:

$$L(M) = \{w \in \Sigma^* | \hat{\delta}(S, w) \cap E \neq \emptyset\}$$

In the next couple of definitions, we will introduce the concept of an ϵ-NFA. ϵ-NFAs differ from NFAs in that they allow epsilon transitions. Transitions labeled with ϵ can be traversed without reading a symbol from the input word. ϵ-NFAs are easier to work with, while still being equivalent to regular NFAs. Most importantly for our purposes, it is much easier to combine ϵ-NFAs together. This allows us to construct the union or the concatenation of two regular languages. ϵ-NFAs are the final concept before we can define Block Automata, which are used in the RegexToNfa exercise. All the definitions and theorems related to ϵ-NFAs are due to [HMU06].

Definition 2.14 (ϵ-NFA)**.** An ϵ-NFA M is defined by the tuple $M = (Z, \Sigma, \delta, z_0, E)$, where $\delta : Z \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Z)$ is the transition function and Z, Σ, z_0, E are defined as per Definition 2.11.

Definition 2.15 (Epsilon Closure)**.** For an ϵ-NFA $M = (Z, \Sigma, \delta, z_0, E)$, we define the function $eclose : Z \to \mathcal{P}(Z)$ recursively.

- **Base case:** $z \in eclose(z), z \in Z$
- **Recursive case:** if $z' \in eclose(z)$, then $\delta(z', \epsilon) \subseteq eclose(z)$

Definition 2.16 (Language of an ϵ-NFA)**.** Given an ϵ-NFA $M = (Z, \Sigma, \delta, z_0, E)$, we define $\hat{\delta} : Z \times (\Sigma^* \cup \{\epsilon\}) \to \mathcal{P}(Z)$ recursively as:

- **Base case:** $\hat{\delta}(q, \epsilon) = eclose(q), q \in Z$
- **Recursive case:** Suppose $w = xa$, where $x \in \Sigma^*, a \in \Sigma$. Let $\{p_1, p_2, \ldots, p_k\} \coloneqq \hat{\delta}(q, x)$, where $q \in Z$. Next, let $\{r_1, r_2, \ldots, r_m\} \coloneqq \bigcup_{i=1}^{k} \delta(p_i, c)$. Then

$$\hat{\delta}(q, w) = \bigcup_{j=1}^{m} eclose(r_j)$$

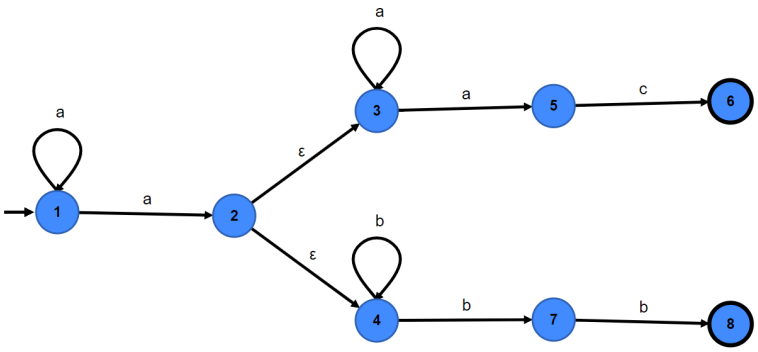

Figure 2.3: An ϵ-NFA

The language L of M is given by:

$$L(M) = \{w \in \Sigma^* | \hat{\delta}(z_0, w) \cap E \neq \emptyset\}$$

Example 2.17. The ϵ-NFA $M = (Z, \Sigma, \delta, z_0, E)$ illustrated in Figure 2.3 accepts the word *aabb* because *there exists a path*, $1 \to 1 \to 2 \to 4 \to 7 \to 8$, starting from state 1 and leading to a final state.

It holds that $L(M) = L(G)$, where G is the grammar from Example 2.6. This also make it equivalent to the DFA from Figure 2.1 and the NFA from Figure 2.2.

Theorem 2.18 (Rabin, Scott)**.** If a language is accepted by an NFA, then there exists a DFA that accepts the same language.

Proof. see [Sch92, p. 32–33] □

Theorem 2.19. Let G be a regular grammar. Then there exists a NFA M with $L(G) = L(M)$.

Proof. see [Sch92, p. 35–36] □

Theorem 2.20. A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA.

Proof. see [HMU06, p. 79] □

Definition 2.21 (Regular expression)**.** Regular expressions are defined recursively, as follows:

- $\emptyset$ is a regular expression
- ϵ is a regular expression
- For every $a \in \Sigma$, a is a regular expression
- If α and β are regular expressions, then so are $\alpha\beta$, $(\alpha|\beta)$ and $(\alpha)^*$.

The language $L(\gamma)$ of a regular expression γ is defined recursively as follows:

- If $\gamma = \emptyset$, then $L(\gamma) = \emptyset$.
- If $\gamma = \epsilon$, then $L(\gamma) = \{\epsilon\}$.
- If $\gamma = a,\ a \in \Sigma$, then $L(\gamma) = \{a\}$.
- If $\gamma = \alpha\beta$, then $L(\gamma) = L(\alpha)L(\beta)$ (the product of $L(\alpha)$ and $L(\beta)$).
- If $\gamma = (\alpha|\beta)$, then $L(\gamma) = L(\alpha) \cup L(\beta)$.
- If $\gamma = (\alpha)^*$, then $L(\gamma) = L(\alpha)^*$.

$\mathcal{R}$ is the set of all regular expressions. For ease of notation we let $\alpha^+ := \alpha\alpha^*, \alpha \in \mathcal{R}$.

Theorem 2.22 (Kleene)**.** The set of languages that can be described through regular expressions is exactly the set of regular languages.

Proof. see [Sch92, p. 37–39] □

Theorem 2.23. An Nondeterministic Finite Automaton, a Deterministic Finite Automaton, an ϵ-NFA, a regular expression and a regular grammar all have the same expressive power, i.e. they are equivalent.

Proof. Follows immediately from Theorem 2.10, Theorem 2.19, Theorem 2.18, Theorem 2.20 and Theorem 2.22. □

Example 2.24. Let $\gamma = a^+(a^+c \mid bb^+)$. It holds that $L(\gamma) = L(G)$, where G is the grammar from Example 2.6. This also make γ equivalent to the DFA from Figure 2.1, the NFA from Figure 2.2 and the ϵ-NFA from Figure 2.3.

Definition 2.25 (Equivalence Relation)**.** To every language L one can assign an equivalence relation R_L on Σ^*. It holds that xR_Ly if for every $z \in \Sigma^*$:

$$xz \in L \Leftrightarrow yz \in L$$

This equivalence relation partitions the language L into equivalence classes. The number of equivalence classes is denoted by $Index\,(R_L)$.

Intuitively, two words w_1 and w_2 are in the same equivalence class if for every suffix w_s, the resulting two words (w_1w_s and w_2w_s) are both either accepted or rejected by the language. If w_1 and w_2 are not equivalent, there exists a *differentiating suffix*, which is a word w_d for which w_1w_d is accepted and w_2w_d is rejected or vice versa. If w_1 and w_2 are equivalent, there exists the *language of suffixes*, which contains all suffixes w_s for which w_1w_s (and implicitly w_2w_s) is accepted. The R_L relation is well defined for all types of languages. Equivalence classes are important for us because they are closely linked to minimal DFAs. They also provide a way to check if a language is regular.

Remark. Let $M_1 \in \mathcal{D}$ be a DFA with $L := L(M_1)$. Now let $M_2 \in \mathcal{D}$ such that $L(M_2) = L$, i.e. M_1 and M_2 recognize the same language.
There exists a function $minimize : \mathcal{D} \to \mathcal{D}$ with the properties listed below. Let $M' = minimize(M_1)$.

- $L(M') = L$, i.e. the minimization function preserves the recognized language.
- $minimize(M_1) = minimize(M_2)$, i.e. the minimization function is invariant with respect to DFAs which recognize the same language.
- Let Z_{M_2} be the states of M_2, $Z_{M'}$ the states of M'. Then $|Z_{M'}| \leq |Z_{M_2}|$, i.e. M' has the smallest state count of any automaton that recognizes L.
- $|Z_{M'}| = Index\,(R_L)$.

There exists a polynomial algorithm to calculate the function *minimize*. For further information about the algorithm as well as a detailed proof, see [Sch92, p. 42–48].

We introduce the following terminology to deal with the exercise type RegexToNfa and the associated Block Automaton. Intuitively, a Block Automaton is an ϵ-NFA which in addition to the normal states can have another type of state, the macro state. Visually, a macro state is depicted as a rectangle (block). This kind of state stands for a whole subautomaton which accepts the language given (as a regular expression) by its label. The subautomaton of a macro state is also referred to as the *definition* of the macro state. Transitions coming into the macro state are considered to connect to its initial state, whereas transitions coming out are considered to come out from every final state of the macro state. In particular, a transition in a Block Automaton can correspond to multiple transitions in the equivalent ϵ-NFA. For the following it is simpler if one imagines the macro state to be an automaton as described above. The only problem left are the final states:

- If the macro state has no outgoing transitions or only self transitions, then its final states are also final states of the Block Automaton.
- Otherwise the final final states of the macro state do not carry over as final states to the entire Block Automaton.

Definition 2.26 (Block Automaton)**.** A Block Automaton is a tuple $M = (Z, T, \Sigma, \delta, z_0, E)$ where:

- Z is the finite set of states.
- $T \subseteq (Z \times \mathcal{R})$ is the finite set of macro states which are stored as a tuple of the state and their respective label (in the form of a regular expression). Additionally $|T \cap (\{z\} \times \mathcal{R})| \leq 1, \forall z \in Z$, i.e. each state has at most one label.
- $z_0 \in \{z \in Z \mid \nexists \gamma \in \mathcal{R}, (z, \gamma) \in T\}$ is the initial state. A macro state cannot be the initial state.
- $E \subseteq \{z \in Z \mid \nexists \gamma \in \mathcal{R}, (z, \gamma) \in T\}$ is the set of final states. A macro state cannot be a final state.
- $\delta : Z \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Z)$ is the transition function.
- Σ is the alphabet.

Definition 2.27 (Language of a Block Automaton)**.** Let $M_{init} = (Z, T, \Sigma, \delta, z_0, E)$ be a Block Automaton and

$$T_m = \{(z, \gamma, M) \in T \times \mathcal{D} \mid L(M) = L(\gamma)\}$$

be the set of macro states of M_{init} combined with a DFA that recognizes the same language as the regular expression of the macro state. Also let $Z_{simple} = \{z \in Z \mid \nexists \gamma \in \mathcal{R}, (z, \gamma) \in T\}$ be the set of non-macro states of M_1 and $Z_{macro} = Z \setminus Z_{simple}$ the set all the macro states. Furthermore, let M_z be the DFA corresponding to macro state $z \in Z_{macro}, \exists \gamma \in \mathcal{R}, (z, \gamma, M_z) \in T_m$.

We define $L(M_{init})$ to be $L(M')$, where $M' = (Z', \Sigma', \delta', z_0', E')$ is an ϵ-NFA with:

- $Z' = Z_{simple} \uplus \left(\biguplus_{(z,\gamma,M) \in T_m} Z_M \right)$ where Z_M represents the states of the automaton M. That is, Z' contains all the states of the Block Automaton except the macro states but additionally has all the states of the DFAs that describe the regular expressions of the macro states.
- $\Sigma' = \Sigma$
- $z_0' = z_0$
- Let $M_e = \{M \in \mathcal{D} \mid \exists \gamma \in \mathcal{R}, \exists z \in Z, \forall c \in \Sigma \cup \{\epsilon\}, (z, \gamma, M) \in T_m \text{ and } \delta(z, c) \subseteq \{z\}\}$, the equivalent DFAs of the macro states that have no outgoing transitions or have only self transitions. Then $E' = E \uplus \left(\biguplus_{M \in M_e} E_M \right)$ where E_M represents the final states of the automaton M.
- Let:

$$\begin{aligned}
S_{delta} &= \{(z, z_0) \mid z \in Z_{macro}, z_0 \text{ is the initial state of } M_z\} \\
E_{delta} &= \{(z, E) \mid z \in Z_{macro}, E \text{ is the set of final states of } M_z\} \\
\delta_{in} &= \{(s, c) \to \{z_0\} \mid s \in Z_{simple}, \exists z \in Z_{macro}, z \in \delta(s, c) \text{ and } (z, z_0) \in S_{delta}\} \\
\delta_{out} &= \{(e, c) \to \{t\} \mid t \in Z_{simple}, \exists z \in Z_{macro}, \exists E, t \in \delta(z, c), (z, E) \in E_{delta} \text{ and } e \in E\} \\
\delta_{both} &= \{(e, c) \to \{z_0\} \mid \exists z, z' \in Z_{macro}, \exists E, (z, z_0) \in S_{delta}, (z', E) \in E_{delta}, e \in E \text{ and } z \in \delta(z', c)\} \\
\delta_{sim} &= \{(s, c) \to \{t\} \mid t \in \delta(s, c), s \in Z_{simple} \text{ and } t \in Z_{simple}\}
\end{aligned}$$

  δ_{in} represents the transitions for which only the target state is a macro state, δ_{out} the ones for which only the source state is a macro state, δ_{both} the ones for which both source and target state are macro states and δ_{sim} the transitions which are between non-macro states. Then:

$$\delta' = \left(\bigcup_{(z,\gamma,M) \in T_m} \delta_M \right) \cup \delta_{in} \cup \delta_{out} \cup \delta_{both} \cup \delta_{sim}$$

Where the first term represents the transitions inside the macro states.

Since $L(M_{init}) \coloneqq L(M')$ (Definition 2.27), the Block Automata have exactly the same expressive power as ϵ-NFAs. They are essentially syntactic sugar for ϵ-NFAs.

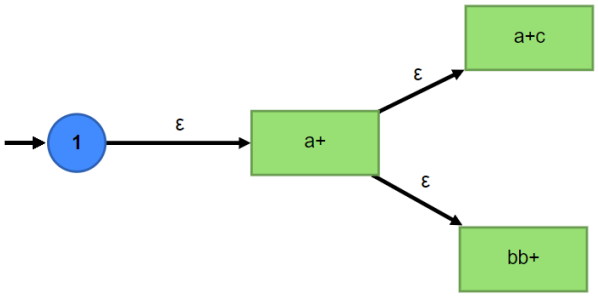

Figure 2.4: A Block Automaton

Example 2.28. The Block Automaton M illustrated in Figure 2.4 accepts the word *aabb* because *there exists a path*, $1 \to a^+ \to bb^+$, starting from state 1 and leading to a final state (inside bb^+). The final states of the macro state bb^+ are final states for the whole automaton, because bb^+ has no outgoing transitions.

It holds that $L(M) = L(G)$, where G is the grammar from Example 2.6. This also make it equivalent to the DFA from Figure 2.1, the NFA from Figure 2.2, the ϵ-NFA from Figure 2.3 and the regular expression from Example 2.24.

The Levenshtein Distance provides a metric for the similarity of two words by counting the number of insertions, deletions and replacements (of symbols) necessary to transform one word into the other. The smaller the distance, the more alike the words are. We use this metric in the grading function of the DescriptionToRegex exercise.

Definition 2.29 (Levenshtein Distance [Lev66])**.** Let a and b be two words, with lengths $|a|$ and $|b|$ respectively. The the Levenshtein distance is given by $lev_{a,b}(|a|, |b|)$ where

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + \mathbb{1}_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $\mathbb{1}$ is the indicator function. To simplify the notation we let $lev(a,b) \coloneqq lev_{a,b}(|a|, |b|)$.

2.2 Structure of Automata Tutor

Automata Tutor is an online learning platform created in [DWWA15]. Developed on the Lift Framework (more information on the configuration of such a project in [Dal13]), it contains a front end and a back end. As mentioned in Section 1.2, Automata Tutor contains numerous exercise types from the field of theoretical computer science. The creation of a new exercise type follows a strict pattern in both the front end and the back end, which are discussed in the following sections.

2.2.1 The Front End

The front end of the project is written in the Scala programming language. This part of the program is responsible for rendering the web pages and handling the database. The web pages range from the log in page, to the courses pages (including pages to create and attend courses, and pose problems), practice

problem list and much more. This thesis is concerned with the three views of an exercise type, namely the create, edit and solve view. The database stores information about the problems that have been posed, the courses that are available, the user base, the solution attempts of the students and more. Since developing a new exercise type does not involve changing the database schemata it will not be discussed any further.

There are a couple of files which need to be created or modified in order to add a new problem type. These are located in different packages and are mostly Scala files, but there are also some HTML and Javascript files. In the following we will go through the steps necessary to create a new exercise type. Please bear in mind that a Scala object is not an instance of a class, as conventionally the case. An object, which needs to be defined separately, is an optional companion to a class with the same name an contains the static behavior of that class.

First, we need to create a Scala file in the `com.automatatutor.model` package. That file contains an object, responsible for static behavior, and a class, containing attributes of the exercise type which need to be stored in order to pose the problem to a student. Depending on the specific exercise, these attributes store a description, an automaton, a regular expression etc. More general attributes as for example a short and long description are handled by a base class. The newly created class must also contain some methods, mostly getters and setters.

Secondly, some modifications need to be made to the `SolutionAttempt.scala` file in order to save the last try of the student. This enables Automata Tutor to restore the previous attempt if an exercise is revisited. As such, we need to create a class which has attributes that can store the input of the student. These are often text fields, because even automata can be saved as text using XML encoding. Then we need to add a companion object whose only function is to retrieve the actual solution attempt instance.

Next, we need to create three HTML files, one for each view of the problem. The three views are: the create and the edit view, also known as *instructor views* and the solve view, also known as *student view*. These are stored in the `webapp` directory in a folder with the name of the exercise type. The three files are usually named `create.html`, `edit.html`, `solve.html`. The HTML files present the static parts of the page and include Lift injection points for the dynamic behavior. These injection points are then used in the Snippet classes presented further down. For the exercise types heavily reliant on Javascript it may be useful to create an additional HTML file that only includes all the necessary Javascript files and libraries as well as the CSS formatting.

In the next step we are required to create a class in the `com.automatatutor.snippet` package which inherits from the `ProblemSnippet` trait (the equivalent of an interface in other programming languages). As a result the class needs to contain four methods: `renderCreate`, `renderEdit`, `renderSolve` and `onDelete`. The `onDelete` method hardly varies between exercise types and won't be discussed any further. The first three functions are each responsible for creating the input fields and processing the input of the user in their view (defined by the afore mentioned HTML files). The processing step may involve sending information to the back end for evaluation, storing the solution attempt (in the case of the student) or creating a new exercise and checking it for validity (in the case of the instructor). In order to send and receive data from the back end, we need to access methods of the `SOAPConnection` class.

The file `SOAPConnection.scala` facilitates the connection to the back end. Its methods package the parameters in XML format and call functions in the `Service.asmx` file of the back end. For a new exercise type it is most likely necessary to create a new method in `SOAPConnection` as well as a new function in `Service.asmx` and set the former to call the latter.

Lastly, we need to add the new exercise type to the mappings in `Problem.scala` and `Boot.scala` and restart the front end.

2.2.2 The Back End

The back end is written in C#. Its main attributions are grading exercises, checking validity and generating exercises. It uses a combination of the Microsoft Z3 Solver [DMB08] and custom-defined functionality to deal with formal languages. For example, the exercises related to regular expressions as well as Description to DFA and a few others use Z3 whereas the PDA and TM exercises use a custom-made implementation.

The file `Service.asmx` represents the API of the back end. Grading functions, e.g. `ComputeFeedbackRegexp`, return a tuple consisting of two XML elements. The first one is the grade and the second one is the feedback. Validity check functions, e.g. `CheckRegexp`, are very similar but only return the feedback to the caller (in

our case the front end). It is recommended to keep the functions in here as short as possible and delegate the computation.

Chapter 3

Expanding the Support for Regular Expressions

We now present four new exercise types added to the Automata Tutor teaching platform around the concept of regular expressions. This chapter will present the exercise types from a student's and instructor's point of view and describe their grading function.

3.1 Words in Regular Expression

Designed as an introduction to the topic of regular expressions, the WordsInRegex exercise type has the goal of familiarizing students with the semantics of regular expressions without requiring much creative input. In this exercise, given a regular expression γ, the student must give words for both of the following types: words accepted by γ and words rejected by γ.

Instructor's view The instructor view of WordsInRegex (Shown in Figure 3.1) is a form containing the following:

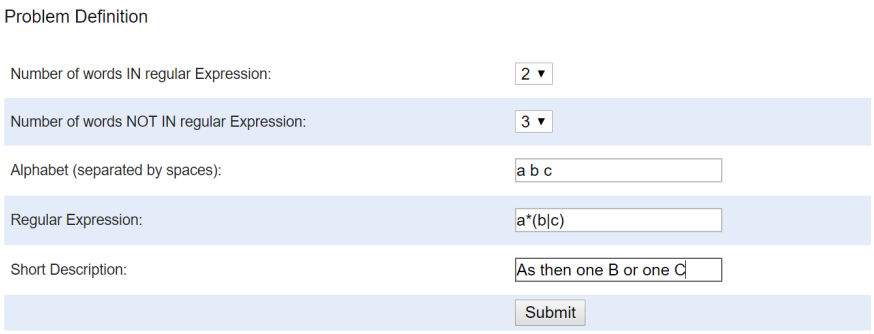

Figure 3.1: Create and edit view of the WordsInRegex exercise type

- A drop-down selector to specify how many words that are accepted by the regular expression should the student provide.
- Another drop-down for the number of rejected words.
- A field for the alphabet of the regular expression.
- A field for the regular expression itself.

- A field for a short description. This will appear if multiple exercises are listed on a page.

After submitting, the expression is checked for syntax errors. If errors are found, a feedback is given to the instructor notifying him about what is wrong. If no errors are found, the exercise is saved in the database.

Student's view The student is given the problem statement in the form of the regular expression, the alphabet and a sentence describing how many words he or she should provide. The sentence has the following format: *Give X words that the regular expression recognizes and Y words that the regular expression doesn't recognize!*, where X and Y are the values selected by the instructor when the problem was created. The problem statement is followed by a corresponding number of input fields for the accepted and rejected words. To insert the empty word the field must be left empty.

To submit an answer, the student has to click on the *Submit* button just below the input fields. This then sends the information to the back end and grades the attempt. This part is the same for all exercise types.

Grading Each word scores equal points, e.g. if the student has to give two words that are recognized by the regular expression and three that are not, then each correctly entered word is worth one fifth of the total points available for the exercise. To check if a word is accepted by the regular expression, the standard functionality of the `Regex` C# class is used. It is not necessary to convert the regular expression to an automaton. The grading mechanism also takes into account if a word has been entered multiple times. In that case, only its first appearance is eligible for points. Moreover, if the same word is used as both accepted and rejected, no points are awarded for the word. This last measure insures that a student cannot get half of the total points just by entering the same word as recognized and not recognized.

3.2 Description to Regular Expression

A good way to elaborate on the knowledge students acquire in the WordsInRegex exercises is the DescriptionToRegex exercise type. This provides the student with a description of the formal language in plain English and they should provide an equivalent regular expression.

Instructor's view Similar to Figure 3.1, the instructor's view for DescriptionToRegex includes input fields for the regular expression, alphabet and a short description. Additionally it has fields for:

- equivalent expressions and
- a long description.

The *long description* is to be given in plain English and should provide enough information to create a regular expression from it. The *equivalent expressions* should describe the same language as the main regular expression and have the aim of allowing for a fairer grading.

Once a problem is submitted, it is checked for validity. This makes sure that the equivalent expressions provided by the instructor are indeed equivalent to the main regular expression. If this is not the case, appropriate feedback is given. Otherwise, the problem is stored into the database.

Student's view The student view consists of a description of the regular expression syntax, a problem statement, which corresponds to the *long description* given by the instructor, and an input field, to write out the solution.

Grading As mentioned before in the instructor views, we store a list of equivalent regular expressions in an attempt to grade the exercise more fairly. Intuitively, we want to see how far off the submission is from a correct solution. For this we use the Levenshtein distance (Definition 2.29), a metric for the similarity of two words.

The grader calculates the smallest Levenshtein distance between the student's submission and one of the equivalent regular expressions given by the instructor. If the submission is equivalent, full points are awarded, regardless if the instructor specified it as an equivalent expression. If the submission is not equivalent, there is a 20% score deduction per unit of distance, with no points being awarded if the submission is not syntactically correct. More formally, let E be the set of equivalent regular expressions, including the main regular expression, w be the student submission and g_{max} the maximum grade. Then, using the notation of Definition 2.29, the formula for the grade is the following: $g_{max} \cdot (1 - 0.2 \cdot \min_{a \in E} lev(a, w))$.

An alternative grading method would have been to transform both the submitted regular expression and the one provided by the instructor into automata and use the existing grading method. We found, however, that similarity on the automata level does not necessary imply the same level of similarity on the regular expression level. The implemented method may not be as complex as the afore mentioned automata grader but acts on the same level, in the sense that no transformations are required. One downside of this approach is that there is more emphasis on the syntax than the semantics. As a result, it is more forgiving when it comes to small errors but may behave poorly when the submission is similar to the language but written in a way that the instructor didn't foresee.

3.3 Equivalence Classes

The exercise type EquivalenceClasses addresses the eponymous topic of regular languages (Definition 2.25). This is a more advanced subject and is not directly tied to regular expressions. The only thing connecting them is that the language needs to be given in the form of a regular expression. Therefore, it can be extended for other representations of regular languages, such as grammars and automata.

Instructor's view As before, the instructor's view consists of a number of fields. Similarly to Figure 3.1, there is a field for the alphabet, one for the regular expression and one for the short description. Additionally there is a selector which allows the instructor to choose what kind of problem he or she wishes to create. There are two options:

- Equivalency between two words and
- Provide words from the same equivalence class.

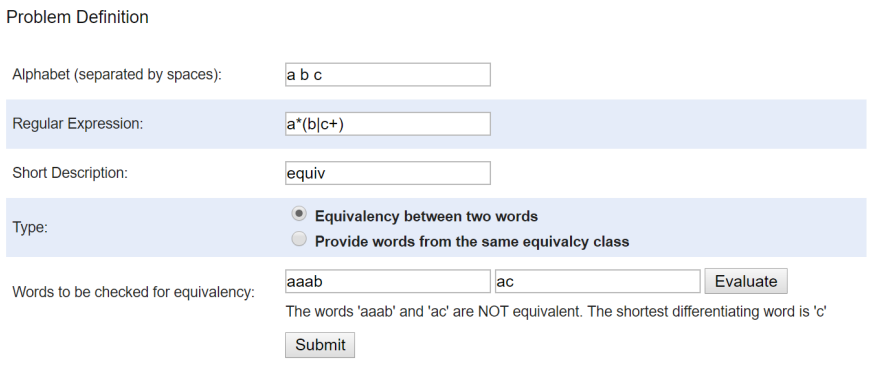

Figure 3.2: Create an EquivalenceClasses exercise

The first one requires two additional words be provided by the instructor. We'll call this variation of the exercise *type-1*. The students will need to decide whether the two words are equivalent and justify their choice. Next to the two input fields is an *Evaluate* button to quickly give the instructor the option to check whether the two in words are equivalent. In addition to assessing the equivalence, the shortest differentiating word (as seen in Figure 3.2) or the language of suffixes is given, depending on the case.

The second option asks for another word, the *representative* of the class. Students will then either have to write the lexicographically shortest word of the class (*type-2*) of the representative or give a number of words from the same class (*type-3*).

Student's view There are multiple solve views, depending on the type of the exercise. However, all of them display the regular expression and the alphabet. For the *type-1* exercise, the two words that are to be checked for equivalence are shown together with a selector. The selector allows the student to choose if he or she believes the words are equivalent. After a selection is made, an input field is revealed to allow the student to give a justification. This takes the form of the language of suffixes, in the case of equivalence, and a differentiating word otherwise.

For the *type-2* exercise, the representative is displayed along with an input field. The student is to write the lexicographically smallest word from the class of the representative. The *type-3* exercise differs from *type-2* by having multiple input fields.

Grading

type-1 This variation has the most complex grading method because it has to handle multiple cases. It gives a third of the points (rounded down to the nearest integer) for a correct assessment of equivalence. The rest of the points have to be obtained from the justification. Let w_1 be the first word and w_2 the second.

In the case of equivalency, the justification needs to be the language of suffixes of the equivalence class. The submitted language, in the form of a regular expression, is converted to a DFA, compared to the correct language of suffixes and graded by the `DFAGrading.GetGrade()` function. To obtain the correct language of suffixes, we transform the regular expression given by the instructor into a DFA M and change its initial state to $\hat{\delta}(z_0, w)$, where z_0 is the initial state of M and $w \in \{w_1, w_2\}$. If the grade given by `DFAGrading.GetGrade()` is greater than one third of the points, we take that to be the grade. If the answer is not correct, we also display the feedback from the afore mentioned function.

In the case the two words are not equivalent, we check whether the differentiating word w_d given by the student is correct. If this is case, full points are awarded. Otherwise, no additional points are awarded besides the assessment. The feedback, shown only if the submission is incorrect, indicates whether both $w_1 w_d$ and $w_2 w_d$ are accepted or rejected by the language.

type-2 In this exercise type the student must give the lexicographically smallest word that is equivalent to the representative. If the student submits the required word, full points are awarded. If the submitted word is in the same equivalence class as the representative, but longer, 66% of the points are awarded, rounded down to the nearest integer. If the submitted word is in the same equivalence class and has the same length as the representative, but isn't the lexicographically smallest one, 80% of the points are awarded. Finally, no points are awarded if the submitted word is in a different equivalence class or over a different alphabet. The feedback indicates which case has been applied.

type-3 The grading for this variation of the EquivalenceClasses exercise is very similar to the one used in Section 3.1. Equal points are awarded for each word, the total being rounded up to the nearest integer.

3.4 Regular Expression to NFA

The exercise type RegexToNfa is the most complex from a technical point of view since it includes a newly developed graphical user interface (GUI) for displaying Block Automata, described in Chapter 4. The object of this exercise is to convert a regular expression into an equivalent Block Automaton.

RegexToNfa exercises form the connection to other ways of representing regular languages, namely finite automata. The concept of transforming between equivalent objects (grammars, finite automata, regular expressions) is key in the field of theoretical computer science since each different representation has its strong suits. One may be preferred for clarity of expression and another for the efficiency of operations. This

is also what Automata Tutor uses behind the scenes to grade and provide feedback for most exercise types. For example, in the previous section we have presented the grading methods of EquivalenceClasses. For every variation of that exercise we transformed the regular expression given by the instructor into an automaton to be able to easily compute equivalence classes. Therefore, transformations are a very important tool with highly practical use cases.

Instructor's view The instructor's view for this exercise type consists of only three input fields, one for the alphabet, one for the regular expression and one for the short description. Upon clicking submit, some basic syntax checks are being performed and, if passed, the problem is stored into the database.

Student's view The solve view of RegexToNfa consists of the problem statement, containing the regular expression and a canvas. The canvas serves an input field for the Block Automaton. Instructions on how to operate it and the functionality it provides are covered in the next chapter. The main idea is that macro states can be used in order to define subsections of the Block Automaton. Thus a macro state has a label, in the form of a regular expression, and a definition, in the form of a (block) automaton. The *Submit* button located below the canvas exports the automaton in XML format to the back end for grading.

Grading The thought behind the grading formula is that students should not be punished for skipping intermediate steps if they are able to come up with a correct solution. In that spirit, equal points are awarded for each definition of a macro state and the whole expression. Previously defined macro states that are no longer used in the automaton are not considered. This way, a student can obtain a maximum score without using any macro states, i.e. without any help of defining subsections of the automaton.

We refer to a macro state as being correct if its definition is equivalent to its label. It is worth mentioning, that in the grading of a particular expression r, be it the whole regular expression or just a macro state, the macro states used to define r are taken at face value. That means, the macro states are assumed to be correct, and their actual definition is not relevant for the grading of r. They will, however, be checked later on in the grading process.

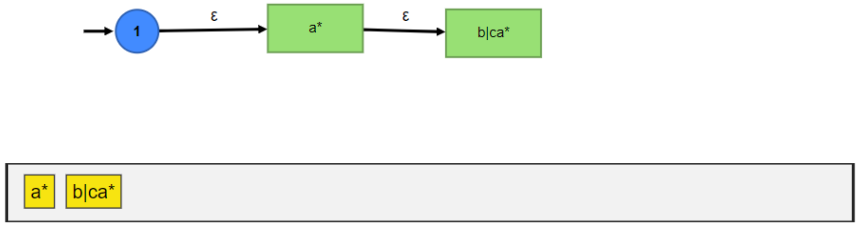

Figure 3.3: Block Automaton for the expression $a^*(b|ca^*)$

For the automaton in Figure 3.3, two different macro states are being used. We can see that by the fact that there are two yellow blocks in the toolbar. This makes for three expressions that need to be defined: the entire expression, $a^*(b|ca^*)$, and the two macro states, a^* and $b|ca^*$. Each of them is worth a third of the total points. From the figure we can see that the entire expression is defined correctly but we cannot say if the macro states are correct. Thus, the submission will be awarded at least one third of the points. Would, for example, the macro state a^* be defined correctly and $b|ca^*$ not, then the submission would get two thirds of the score.

Chapter 4

Automaton GUI

A custom graphical user interface (GUI) has been developed to facilitate an effortless input of a Block Automaton. The GUI takes advantage of the Javascript library `d3.js` [BOH11] to allow interaction with the automaton. Based on Ross Kirsling's *Directed Graph Editor* [Kir13], numerous additions have been made in order to support different types of automata. One can create states, transitions, macro states and define the macro states by creating another Block Automaton inside of them.

4.1 User viewpoint

This section will describe how to operate the GUI and the few restrictions it imposes. First we define some terminology. As mentioned during Section 3.4, a macro state has a label, in the form of a regular expression, and a *definition*, in the form of a (block) automaton. A macro state is said to be *expanded* if it's automaton is displayed on the screen. At all times, a single automaton is displayed in the interface. The initial automaton is called the *main automaton*, i.e. the automaton that is displayed when no macro states are expanded. We'll refer to the *current context* as being the currently expanded macro state or the main automaton if no macro states are expanded. Let s_1 be a macro state labeled $a|b^*$ and s_2 be a macro state labeled b^*. Furthermore, assume s_2 is part of the definition of s_1. Then s_2 is said to be a *child* of s_1 and s_1 the father of s_2. Generalizing, an *ancestor* of a macro state is defined recursively as its father or any ancestor of its father.

States Macro states are represented as green rectangles and non-macro states as blue circles. The term states will be used to refer to both the graphical representation and the theoretical concept, because there is a one-to-one correspondence. To create a simple (non-macro) state, left-click on an empty area of the canvas, or right-click and select the option `Add state`. To add a macro state, right-click and select `Add block state`. This opens a text field, surrounded by a rectangle, in which the label of the macro state has to be entered. After the desired label has been typed, the enter key needs to be pressed to add the newly created macro state to the current context.

To select a state simply click on it and its color will change to a lighter shade to indicate it has been *selected*.

To remove a state, right click on the state and select the remove state option. This works for both macro and non-macro states. Initial states cannot be removed. If a state is selected, it can be removed by pressing the `backspace` or `delete` key on the keyboard.

A non-final state can be made a final and vice versa by right-clicking on it and selecting `Toggle final`. The same result is achieved by double clicking or pressing the `F` key while a state is selected. Final states are indicated by a black border around the circle. Only non-macro states can be made final. Whether a macro state is final or not is decided by its outgoing arrows Definition 2.27.

Transitions Transitions are represented as arrows between states. These arrows are labeled with symbols of the alphabet and ϵ. An arrow label may contain multiple symbols. In that case, the arrow stands for

multiple transitions. It is also possible that more than one arrow is part of the same transition. As shown in Figure 4.1, arrows define an injective function while the transition function for Block Automata is not necessary injective (e.g. $\delta(1, a)$). From now on, the term *arrow* will denote the graphical element while the term transition is reserved for the theoretical concept.

When hovering over a state, an overlay (referred to as a *halo*) containing the symbols of the alphabet and ϵ is shown, as can be seen in Figure 4.1 around state 2. The symbols are the starting points for creating a transition. To create a new arrow, left-click on a symbol, drag to another state and release. While dragging and hovering over a state, that state will enlarge to indicate that if the mouse button would to be released, that state would be the target of the arrow. Also, while dragging, a dotted arrow is drawn from the source state to the current mouse position. If the mouse button is released over empty space, the arrow is discarded.

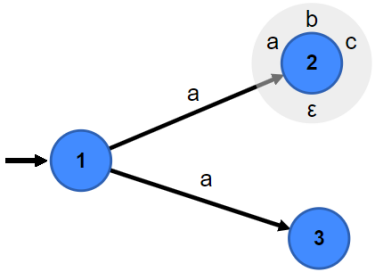

Figure 4.1: Injective arrows vs transitions

Arrows can also be moved around. Let l be an arrow from state s to state t, and c be a symbol of its label. If c is clicked, it is removed from the label of l. At this moment, the interface is in the same situation as when dragging from the symbol c of the halo of s.

One can also remove symbols from the arrows' labels as well as entire arrows by right clicking and selecting `Remove label` or `Remove entire edge`.

Macro states Macro states have additional features. Upon double click or right-click the macro state is expanded. One can also expand a macro state from the toolbar. This presents the user with a bounding box, as seen in Figure 4.2. Inside this bounding box one can define the macro-state, i.e. input the automaton of the macro state. All interactions (regarding states and arrows) happen in the same way as for the main automaton. Macro states can be nested in one another, arbitrarily deep.

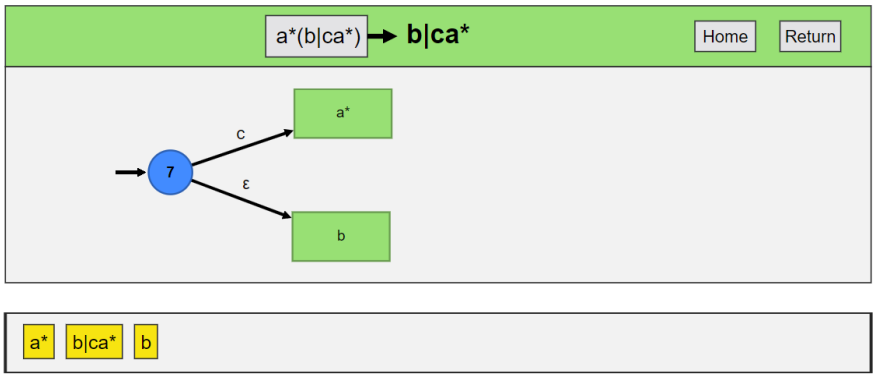

Figure 4.2: Expanded macro state

The header of the macro state presents the so called *stack trace*, which shows the ancestors of macro states going back from the expanded state. Clicking a rectangle of the stack trace expands that macro state. If the stack trace is longer than there is room on the header, the more distant entries are not shown. To the right are two buttons, labeled `Home` and `Return`. The former displays the main automaton and erases the entire stack trace, whilst the latter expands the father state.

The GUI also provides *sync* functionality. This means that all the macro states with the same label have the same definition at all times. The state numbers differ from one macro state to another but the graphs will be isomorphic. This feature imposes the following restriction on the nesting of macro states. It is not allowed to add a macro state labeled l_{new} to the current context if the current context or any other macro state having the same label as the current context has an ancestor labeled l_{new}. This measure is taken to prevent the infinite loop that would be caused if a macro state could be its own ancestor. Furthermore, all macro states created by the user have a common ancestor, called the *root*. Its label is the entire regular expression and its definition is the main automaton.

Figure 4.3 presets an invalid operation. An arrow from a rectangle r_1 to another rectangle r_2 in the diagram signifies that the macro state labeled r_1 is the father of the macro state labeled r_2. Dotted lines represent pending actions. For brevity we write state x when in fact we mean the macro state labeled x. The user tries to add the state b (depicted as the dotted red rectangle) as a child of the state c. This is not allowed, because on a different branch, another state b is the ancestor of another state c. One must remove the state c on the left branch before adding the red macro state.

Toolbar Also to be observed in Figure 4.2, the toolbar resides on the bottom of the GUI. It displays the macro states that have already been used. If a rectangle in the toolbar is double clicked, the corresponding macro state is expanded. The same can be achieved by right clicking and selecting `Edit`. If a macro state is expanded from the toolbar, no stack trace is shown. Additionally, one can also add a macro state from the toolbar to the current context. Macro states cannot be removed from the toolbar, because at the moment there is no way to check if a macro state is in use. The mouse wheel or trackpad can be used to scroll through the toolbar if it is overfilled.

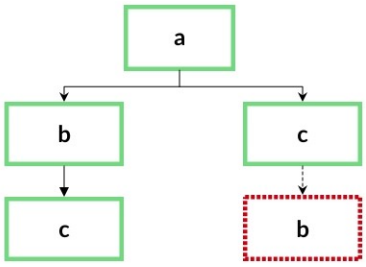

Figure 4.3: Invalid hierarchy

4.2 Implementation

In this section we'll describe some of the features of the GUI from a more technical point of view.

4.2.1 Overview

File structure All the Javascript files of the front end are stored in `src/main/webapp/javascript` directory. The main part of the code, including all the methods for user interaction, the toolbar and the declaration of the context menu is located in the `blockInterface.js` file. The entire file is occupied by a function constructor for the `BlockCanvas` object. The classes `SimpleNode` and `BlockNode` as well as `Links` are defined in the file `blockClasses.js`. The context menu is declared in `blockContextMenu.js`. CSS styling information is stored in `src/main/webapp/stylesheets`, with `proto2.css`, `blockInterface.css` and `blockContextMenu.css` being relevant for the GUI. These six files need to be included in the web page for the GUI to work properly.

Configuration The `BlockCanvas` constructor has a couple of parameters:

1. `container`, a `string` which specifies the identifier of the HTML element, recommended to be a `div`, in which the Scalable Vector Graphics (SVG) canvas should be placed.
2. `dimensions`, an `Array` with two elements, the width and height of the canvas respectively (excluding the toolbar).
3. `deterministic`, a `boolean` with the default value of `false`, specifying if the automaton should be a DFA or NFA.
4. `epsilon`, a `boolean` with the default value of `true`, which specifies if ϵ transitions are allowed. This shows or hides the ϵ at the bottom of the halo, allowing or forbidding epsilon transitions.
5. `blockAutomaton`, a `boolean` with the default value of `true`, which specifies if macro states are allowed.

If the `deterministic` flag is set to `false`, the halo will be displayed as a way to add arrows. If, however, `deterministic` is set to `true`, more restrictions ensue. Notice that is this case, a labeled arrow is equivalent to a transition. The halos are hidden, therefore arrows cannot be added. All arrows are automatically added by the GUI as self-loops and they can be modified by the user. If released while moving, the arrow is added back in the form of a self-transition, not discarded as before. Hence transitions cannot be removed, just edited.

API The API of a `BlockCavas` contains the following functions:

- `clear()`, which clears all data objects.
- `setAutomaton(str)`, which, given the XML representation of an automaton (as a string), sets the displayed automaton to match the one described by the argument. This is used to load a previous attempt, if available.
- `lockCanvas()`, which disables user interaction. This can be used, for example, in the NFA to DFA exercise type, to present the NFA to the student.
- `exportAutomaton()`, which returns the XML representation of the Block Automaton as a string. This is used to send the automaton to the back end for grading.
- `reset()`, which removes all transitions and states except for the initial state.

Additionally, there are the functions `setAlphabet(str)`, `setRegex(str)`, `setEpsilon(flag)`, `setAlphabetArray(arr)` and `exportAlphabet()` which are self-explanatory.

4.2.2 Code Synopsis

In this section we'll provide a brief summary of the main components of a `BlockCanvas` object. The core of the interface is the `d3.js` force simulation. Force simulations are used to represent graph like structures, which makes them the ideal choice for finite automata. We will also present how macro states are expanded and give some details on the toolbar. But first we provide some insight into the different classes used to represent arrows and states.

Classes Figure 4.4 presents the relationships, attributes and methods of the classes used in the GUI. First we have the `Node` class which is used as an interface and represents the states of an automaton. Strictly speaking it isn't an interface, but objects of type `Node` are never used. It provides common attributes like position (`x` and `y`), which are needed for the force simulation, a globally unique identifier (`id`), and flags for final (`reflexive`), initial (`initial`) and macro (`isBlock`) states. `Node` also provides a constructor. The class `SimpleNode`, representing non-macro states, inherits from `Node` and overrides the constructor.

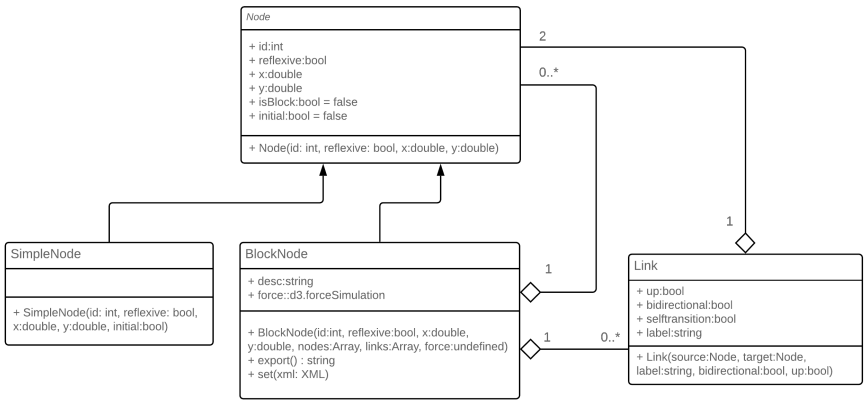

Figure 4.4: Class structure

`BlockNode` also inherits from `Node` but adds more functionality for macro states. A new `desc` field is added for the label of the macro state. Two arrays, `links` and `nodes`, symbolized by the composition relationship, are also added. They are used to store the definition (automaton) of the macro state. Each `BlockNode` instance also gets its own force simulation. Regarding methods, three new are added, including a constructor to deal with the new attributes. The method `export()` returns the XML representation of the definition of a macro state s_m. This includes the definitions of all the macro states which have s_m as an

ancestor. Lastly, `set(xml)`, sets the definition of the macro state to the Block Automaton described by the `xml`.

As touched upon in Section 4.1 and Figure 4.3, macro states are stored in a hierarchical structure, with each macro state having a father, except for the root macro state. One can visualize this structure as a tree rooted at the root macro state. This special macro state is stored in a variable called `root`, with `id` zero, which is never removed or replaced during the execution of the program. In the case the automaton is set or cleared, using the `setAutomaton()` and `reset()` functions respectively, we do not create a new root macro state, we only update the fields of `root`.

A `Link` represents an arrow. Its only method is the constructor and its attributes are:

- `source`, a reference to the source `Node`.
- `target`, a reference to the target `Node`.
- `label`, the label of the arrow.
- `up`, which indicates if the source `Node` has a greater `id` than the target `Node`.
- `bidirectional`, which indicates if there exists another arrow going from `target` to `source`.
- `selftransition`, which indicates if the arrow is a self-loop.

Additionally, there are two objects, `BlockNodeStatic` and `SimpleNodeStatic`, which contain information regarding the size of the circles and rectangles that represent the states, the dimensions of the bounding box for expanded macro states, etc. `BlockNodeStatic` also contains a list of all the macro states, in the form of an `Array` named `BlockNodeStatic.blocksList`.

Force Layout `d3.forceSimulation` requires a list of nodes and optionally a list of links, each in the form of an `Array`. The force layout modifies the position of the nodes in the simulation based on a set of forces that are defined at the start of the simulation. `d3.forceSimulation` expects links to have two attributes named `source` and `target`. The force simulation uses the `x` and `y` attributes of the `Node` class to set the position of the nodes. One can access the position of a node by the same attributes.

```
1  currentContext.force = d3.forceSimulation()
2    .force('link', d3.forceLink().id((d) => d.id).distance(150))
3    .force('charge', d3.forceManyBody().strength(-500).distanceMax(100))
4    .on('tick', tick.bind(this, (width - BlockNodeStatic.maximizedWidth)/2, (height -
         BlockNodeStatic.maximizedHeight)/2 + BlockNodeStatic.headerHeight, BlockNodeStatic.
         maximizedWidth, BlockNodeStatic.maximizedHeight - BlockNodeStatic.headerHeight));
5
6  currentContext.force
7    .nodes(currentContext.nodes)
8    .links(currentContext.links);
9
10 currentContext.force.alpha(.9).alphaDecay(.04).restart();
```

Listing 4.1: Initializing a force simulation

Listing 4.1 presents how one would use a force simulation. The variable `currentContext` represents the current context. First (lines 1 through 4), the force simulation is initialized. Two kinds of forces are defined, a `link` force, which acts on pairs of nodes connected by an arrow, and a `charge` force, which acts on all the nodes. In line 4, a tick function is added. The tick function is called at regular time interval as long as the simulation is active and has the purpose of updating the position of the graphical elements (circles, rectangles and arrows). The arguments passed to the tick function define the dimensions of the bounding box in which the nodes must reside. Next (lines 6 through 8), the nodes and links are added to the simulation. The forces of a simulation decrease over time and need to be *reheated* whenever the layout changes (through the addition/removal of a node or link). More explicitly, all the forces of the simulation are multiplied by a so called *alpha* value, which decays over time eventually reaching zero. The simulation is said to be active if its alpha value is greater than zero. Line 10 reheats the simulation by setting the alpha value and its decay rate and restarting the simulation.

Changing context The current context is changed by expanding macro states. Furthermore, the user may pick any macro state from the stack trace to return to. To facilitate this change of context, the GUI stores all macro states from the stack trace in a stack named `contextStack` while the current context is stored in `currentContext`. Whenever a macro state is expanded, either from the current context or from the toolbar, `currentContext` is pushed onto the `contextStack` and then `currentContext` is changed to the newly expanded macro state. If the `Return` or `Home` buttons are pressed, or if the user clicks on a macro state from the stack trace in the header of the bounding box, the `contextStack` is popped until the popped state s_{pop} corresponds to the requested macro state. s_{pop} is then stored in `currentContext` and expanded.

The context change is handled by the `replaceContext()` function. The same function removes the graphical elements of the previous context, draws the bounding box, stack trace, initializes the simulation of the newly expanded macro state and calls the `restart()` function.

Restart function The `restart()` function, not to be confused with the `currentContext.force.restart()` function, is called whenever the layout of the automaton has changed (as mentioned earlier, through the addition/removal of a node or link). In said function, the nodes and links of the `currentContext` are associated to graphical elements (circles, rectangles and arrows) with a standard `d3.js` data merge. `restart()` also adds the halos and all the event handlers for nodes and links. At the end of the function, `currentContext.force.restart()` is called to reheat the simulation. This arranges the elements neatly on the canvas.

Toolbar The toolbar is displayed at the very bottom of the GUI, as seen in Figure 4.2. It is in fact on a second SVG canvas, just below the first one. The toolbar shows all the macro states used so far. For this to be possible, all macro states are stored in an array named `blocksArr`. The variable `BlockNodeStatic.blocksList` points to the same array. The elements of `blocksArr` are 2-tuples, consisting of a `boolean` and a `BlockNode`. The `boolean` is a flag which indicates if the macro state is the first of its label. Only those macro states are added to the toolbar. In this way no duplicates are displayed. The function constructor `Toolbar()` creates a new toolbar object. Its API consists of a `render()` and `add()` function. The former removes all existing elements from the toolbar, draws the rectangles for the macro states and sets their event handlers, calculates the scrolling bounds and attaches the scrolling handler. The latter adds and additional macro state to the toolbar.

4.2.3 Noteworthy code sections

In this section we'll describe some of the challenges encountered during the implementation and how they were addressed.

Sync All mentioned in Section 4.2.2, all the *similar* macro states (i.e. macro states with the same label) are synced, meaning their definitions are isomorphic. To achieve this, all macro states are stored in `blocksArr` (4.2.2). Each macro state is represented by its own `BlockNode` object. This ensures each macro state has a unique `id`. Syncing the nodes in the definition of similar macro states is not difficult; whenever a new state is added to a macro state s_m, we add a state to all the macro states similar to s_m. Links are more complicated. We make the following remark:

Remark. Let M and M' be the definitions of two similar macro states with state sets Z_M and $Z_{M'}$. Let z_k be k^{th} order statistic of the state set of M and similarly z'_k for M'. Because of the way nodes are synced and the fact that the definitions of the two macro states are isomorphic, there exists an arrow between z_i and z_j if and only if there is an arrow with the same label between z'_i and z'_j, where $i, j \leq |Z_M|$.

Less formally, if we were to renumber the states of similar macro states by their order statistic, we'll get the exact same automaton. This allows us to identify equivalent links in similar macro states. We refer to the k^{th} node as being the node with the k^{th} smallest `id`. Thus, if a new link is added from the i^{th} to the j^{th} node in the macro state s_m, all similar macro states get a new link between their i^{th} and j^{th} node.

Using the above remark we are able to keep similar macro states in sync. The functions involved in this are `syncNode()`, `syncRemoveNode()`, `syncLink()` and `syncRemoveLink()`. The function `buildBlock(desc)`

constructs a new macro state with the label `desc`. If the label was already used, the new macro state is synced with all similar macro states.

An alternative method to provide sync functionality would have been to use the same object for all similar macro states and just reference it whenever used. This would sync similar macro states without any additional work, since we would update the underlying object. There is however an important limitation to be considered. By using references, we cannot add multiple macro states with the same label to the main automaton or any subautomaton (used to define a macro state). This is because we would not have a way of specifying which arrows go to which of the similar macro states, since the `target` attributes of the links would point to the same state. We also wouldn't be able to control their position independently, because they share the position attributes. Because of this serious restriction we have decided to use deep copies for similar macro states instead of references.

Depth first search As described in Section 4.1 and shown in Figure 4.3, the sync feature imposes restrictions on the hierarchy of macro states. To check if a new macro state is valid in the current hierarchy, the function `validBlock()` and its helper function `validBlockHelper()` perform a depth first search (DFS) from the root of the automaton. `validBlock()` returns 1 if the macro state can be added and 0 otherwise.

```
1  function validBlockHelper(block, stack, descAttempt, descFather){
2    stack.push(block.desc);
3    if(block.desc === descFather && stack.includes(descAttempt)){
4      stack.pop();
5      return 0;
6    }
7    let ret = 1;
8    block.nodes.filter(d => d.isBlock).forEach(neigh => {
9      ret &= validBlockHelper(neigh, stack, descAttempt, descFather);
10   });
11   stack.pop();
12   return ret;
13 }
```

Listing 4.2: Validity check on macro state to be inserted

The bulk of the calculations is done by the `validBlockHelper` function (Listing 4.2). The `block` parameter is of type `BlockNode` and is the current node (macro state) of the DFS in the hierarchical tree. `stack` is an `Array` that stores all the ancestors of `block`. `descAttempt` and `descFather` are both `strings`, the label of the macro state to be inserted and the label of the current context respectively. For every node labeled `descFather` we check if the `stack` contains the label `descAttempt`. If it does, the hierarchy is invalid and we return 0. If not we check all the children of `block` and if all of them comply with the restrictions, we return 1, otherwise we return 0. Logical *and* (`&`) is used to compose the return values of the children. Before every return, we pop the stack to reflect the ancestors of the father.

Geometry The `tick()` function makes heavy use of geometry in order to move the graphical elements around the canvas. For the circles and rectangles a simple translation suffices, but in the case of the arrows' labels and bidirectional arrows, the perpendicular bisector needs to be calculated.

The function `perpendicularBisector()` (Listing 4.3) calculates the coordinates of a point $p_{support}$ on the perpendicular bisector. `p1` and `p2` are the endpoints of the line l. They both have an `x` and `y` attribute. `relative` is the distance between $p_{support}$ and the midpoint of l relative to the length of l. `absolute` is the additional distance (on top of the one specified by `relative`) to the midpoint. `locked` is a flag which specifies if $p_{support}$ should always be on the same side of l (with respect to the directed line between `p1` and `p2`). For bidirectional links it is set to true and for the labels of non-bidirectional links it is set to false. `perpendicularBisector()` returns an object containing the coordinates of $p_{support}$ and the slope of the perpendicular bisector.

First, we calculate the square length of l and its midpoint coordinates. This is followed by the slope `m` of the perpendicular bisector and its y-intercept. `dx` specifies the distance between $p_{support}$ and the midpoint on the x-axis. The formula comes from the following calculation. Let r be equal to `relative`, a to `absolute`, d_l to length of l and d to the distance between the midpoint and $p_{support}$ and d_l. Then the following

equations hold:

$$\begin{aligned} d &= d_l \cdot r + a \\ d^2 &= d_x^2 + m \cdot d_x^2 \\ d_x &= \sqrt{\frac{d^2}{m^2+1}} \\ \text{so } d_x &= \sqrt{\frac{d_l^2 \cdot r^2 + a^2 + 2a \cdot d_l \cdot r}{m^2+1}} \end{aligned}$$

In the implementation we ignore the $2a \cdot d_l \cdot r$ part of the numerator because we get a good constant offset even without it.

```
1  function perpendicularBisector(p1, p2, relative, absolute = 0, locked = true) {
2    const distSquared = (p2.x - p1.x) * (p2.x - p1.x) + (p2.y - p1.y) * (p2.y - p1.y);
3    const midpointX = (p1.x + p2.x) / 2;
4    const midpointY = (p1.y + p2.y) / 2;
5    const m = -(p2.x - p1.x) / (p2.y - p1.y);
6    const intercept = midpointY - m * midpointX;
7    let dx = Math.sqrt((distSquared * relative * relative + absolute * absolute)/((m*m + 1)));
8    let supportX, supportY;
9    if(locked){
10     const orient = p1.x > p2.x ? 1 : -1;
11     supportX = midpointX + (orient * m > 0 ? -dx : dx);
12     supportY = (supportX * m + intercept);
13   }
14   else{
15     supportX = midpointX + (m > 0 ? -dx : dx);
16     supportY = (supportX * m + intercept);
17   }
18   return {'x':supportX, 'y':supportY, 'm':m};
19 }
```

Listing 4.3: Calculations for perpendicular bisector

Next, we differentiate between to cases: $p_{support}$ should always be on the same side of l or it should be above it (this actually means it has a smaller y-coordinate than the midpoint, since SVG flips the y-axis). In the former case we enter the `if` statement in line 10. The calculations (lines 10-11) make sure that regardless of the position of the endpoints and the slope of the perpendicular bisector, `supportX` lands on the correct side of `midpointX`. With the x-coordinate of $p_{support}$ set, we use the slope `m` to get the y-coordinate (line 12). In the `else` case the sign of the slope `m` compensates for the position of the endpoints and makes sure the label is above the arrow. Finally, we return the coordinates of $p_{support}$ alongside the slope `m`.

To draw a bidirectional link we use the point $p_{support}$ as an intermediary point between the source and the target node. Then we use the `d3.line()` functions to create a curve which goes through all three points.

Chapter 5

Performance Benchmark

This chapter covers a couple of tests conducted on the RegexToNfa exercise type to see how it handles big automata. The purpose of these tests was to see if the GUI and the back end grading methods are efficient enough for normal use cases.

5.1 Testing methodology

We introduce some terminology. The *count* of a Block Automaton is the number of states of that automaton. We also say a macro state has count c if its definition has count c. Recall that Block Automata can be nested in one another using macro states. We define the *level* of a macro state as the number of its ancestors. For non-macro states it is defined in the following way. Let s be a non-macro state part of the definition of the macro state m and let l be the level of m. Then s has level $l + 1$. In this way, all states part of the same definition are on the same level. Recall that the main automaton is actually the definition of the root macro state. For example, states directly part of the main automaton are on level one, while the states displayed in Figure 4.2 are on level two. The *depth* of a Block Automaton is the maximum level of a state. We also say a macro state contains a state s when we actually mean that the definition of that macro state contains s.

We generated four types of test cases: dense, sparse and random automata, and a few-macros type. These all have a specific hierarchical structure. The root macro state has the label a^{10} and all the macro states on level l have the label a^{10-l}. All the macro states as well as the main automaton have the same count, except for the macro states at the last level. A macro state at level l contains an initial non-macro state and $count - 1$ macro states labeled $a^{10-(l+1)}$. In the case of dense, sparse and random automaton types, the macro states on the penultimate level contain only an initial state with no transitions. In the few-macros case, those macro states contain *count* non-macro states. The last level never contains macro states. This structure of the block automaton allows us to create exponentially bigger automata by adjusting the count parameter for the base and the depth parameter for the exponent.

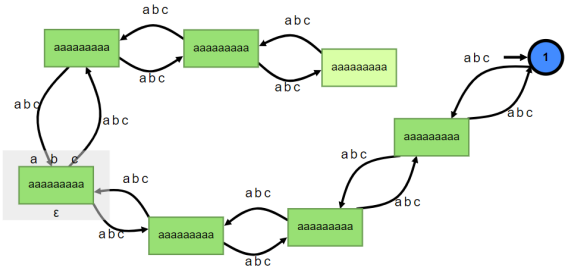

Figure 5.1: Sparse automaton with count eight

Besides the above mentioned hierarchical difference, the test automata only differ in their transition set.

For all test types, the definitions of all the macro states and the main automaton are isomorphic (in the chain and dense case) or very similar (in the random and few-macros cases), except for automata on the last level. The dense automata are complete graphs (i.e. there is a link from every node to every other node). The sparse automata are chain graphs (i.e. each node is connected to the first node with a greater `id`). For those two cases, all links are labeled with all the letters of the alphabet and bidirectional, in an attempt to exclude any back end optimizations designed for DAGs. For the random and few-macros automata, every link is constructed with probability 20%, for the automaton with count 30, or 25% for all the rest. The label of a link is randomly assigned as one symbol of the alphabet and with a probability of 25% epsilon is added to the link label. For the few-macros automata, the last level gets its transitions just as any other level.

To measure the execution time, we have added a timer to the back end grading function, measuring the time from the beginning of the function execution until the function returns. Thus, the time shown in the Table 5.1 represents the time taken by the back end for grading, averaged over five runs. There is a limit on the size of the packets that can be sent between the front end and the back end. We had to increase that limit for our testing purposes, since it turned out to be a bottleneck. For every test instance, we have set the automaton in the GUI using the `setAutomaton()` function and pressed the *Submit* button for grading.

5.2 Results

In the following we examine the results of our performance testing and draw some conclusions on the effect of different properties of the Block Automaton on the execution time of the grading process.

Case	Config.	Count	Depth	States	Blocks	Links	Time (sec)
dense	deep	5	7	10921	5461	27300	17.18
	middle	8	5	5601	2801	22400	8.54
	wide	30	3	1741	871	26100	2.82
sparse	deep	5	7	10921	5461	10920	16.32
	middle	8	5	5601	2801	5600	8.1
	wide	30	3	1741	871	1740	2.54
random	deep	5	7	10921	5461	≈9550	16.39
	middle	8	5	5601	2801	≈7000	8.56
	wide	30	3	1741	871	≈5600	2.75
few macros	deep	7	6	10885	1555	≈9550	5.50
	middle	9	5	5265	585	≈12000	2.27
	wide	40	3	1600	40	≈16000	1.2

Table 5.1: Performance test results

Table 5.1 presents some additional quantities. *States* represents the total number of states, *Blocks* the total number of macro states (including the root) and *Links* the total number of arrows. We picked three different automata configurations for each type (*deep*, *middle* and *wide*). The first configuration has a big depth, meaning there are a lot of macro states, each with its own automaton. The count, however, is small by comparison and this yields simple automata. The last one has few but complex macro states. The second strikes a compromise between the other two. We can see that the total number of states for each configuration is comparable between cases.

Analysis In the dense case we can see that the number of links is kept constant, yet the execution time varies wildly. Moreover, in the few-macros case, the number of links seems to be negatively correlated to the execution time. However, this cannot be the case. For the dense, sparse and random cases the execution time hardly varies for each configuration. We can therefore exclude the number of transitions from the list of factors for high execution time. Next, we turn our attention to the *States* quantity. In the first three cases (dense, sparse and random) it appears to be very much connected to the execution time. Unfortunately

it is also highly correlated to the *Blocks* quantity. The few-macros case shows a different picture. All the configurations are evaluated much faster than in all other cases, despite comparable number of states. This is especially apparent in for the *middle* configuration. We conclude that the number of macro states is the deciding factor for execution time.

We check our supposition in R (Listing 5.1) with a multiple linear regression with *States*, *Blocks* and *Links* as covariates. We get a high level of significance for the *Blocks* covariate, with a p-value of `7.04e-10` for the T-test. Overall, the regression is highly significant with a p-value of `4.413e-11` for the F-statistic. This confirms our hypothesis from the previous paragraph.

```
Call:
lm(formula = time ~ ., data = perf)

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -8.366e-02  2.187e-01  -0.383   0.7120
states       9.958e-05  4.214e-05   2.363   0.0458 *
blocks       2.766e-03  8.279e-05  33.406 7.04e-10 ***
links        2.677e-05  1.160e-05   2.306   0.0500 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3179 on 8 degrees of freedom
Multiple R-squared:  0.9979,  Adjusted R-squared:  0.9972
F-statistic:  1293 on 3 and 8 DF,  p-value: 4.413e-11
```

Listing 5.1: R output

During the tests, we could also asses the responsiveness of the GUI. We noticed that the GUI was not affected by the *States*, *Blocks* and *Links* quantities. It was however influenced by the configuration and case, with the dense and *wide* automaton behaving the poorest. This suggests the responsiveness of the GUI is only impacted by the size of the current context and not the total size of the Block Automaton.

Conclusion Based on the results of the performance testing, we conclude that the RegexToNfa exercise is more than capable of dealing with a normal load scenario. It is unrealistic to assume a student will create an automaton with thousands of macro states. The back end provides a very efficient implementation, in part due to the Z3 Solver [DMB08]. The limitations lie in the package size limit imposed on the communication between front end and back end as well as in the responsiveness of the GUI for large automata.

Chapter 6

Conclusion

Summary In this thesis we have introduced three new exercise types to Automata Tutor and improvements to a fourth one, all related to regular expressions. Each exercise type has its own difficulty, ranging from introductory exercises (WordsInRegex) up to advanced ones (EquivalenceClasses). This allows students to progress through the topic of regular languages and develop an understanding for regular expressions. The exercises WordsInRegex and DescriptionToRegex serve as a foundation in this regard. In the RegexToNfa exercise, they see the connection between regular expressions and other equivalent representation forms. In the EquivalenceClasses exercise, students are introduced to the eponymous concept. We have given an overview of the exercises from a user point of view and explained the grading methods used.

We have developed a graphical user interface for the RegexToNfa exercise, so that students can interact with a Block Automaton. The GUI has then been expanded to support DFAs and NFAs. Thereby it can be used to replace the old GUI for a number of automata exercise types and serves as an alternative for future exercises. It is our hope that the implementation along with the documentation provides a good basis for further development of the GUI to eventually include all automaton types in Automata Tutor.

Future work The transformation from NFA to regular expression has been omitted from this thesis due to time constraints. It involves a new automaton type whose link labels are regular expression. This involves creating GUI support for this new automaton type as well as finding a sensible way to grade such a submission.

We would also like to see the EquivalenceClasses exercise expanded to support multiple definitions of the language. Since equivalence classes are well defined for all formal languages, not only regular languages, any definition of a language (e.g. finite automata, PDAs, Type-2 grammars) would constitute a valid exercise.

Acronyms

API application programming interface. 22, 24

CNF Chomsky normal form. 2

CYK Cocke–Younger–Kasami. 2

DAG directed acyclic graph. 28

DFA Deterministic Finite Automaton. 2, 5, 6, 8, 9, 10, 12, 17, 21, 30

DFS depth first search. 25

GUI graphical user interface. 1, 3, 17, 19, 20, 21, 22, 24, 27, 28, 29, 30

NFA Nondeterministic Finite Automaton. 1, 2, 6, 7, 8, 9, 10, 21, 30

PDA Pushdown Automaton. 2, 12, 30

SVG Scalable Vector Graphics. 21, 24, 26

TM Turing Machine. 2, 12

List of Figures

List of Tables

Listings

Bibliography

[Ajd18] Tohid Ebrahim Ajdari. Extending the tool AutomataTutor with Turing Machines. B.Sc. Thesis, Technische Universität München, 2018.

[Bac18] Christian Backs. Automatische Aufgabenerzeugung für Turingmaschinen. B.Sc. Thesis, Technische Universität München, 2018.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[Dal13] Richard Dallaway. *Lift Cookbook: Recipes from the Community for Building Web Applications with Scala*. " O'Reilly Media, Inc.", 2013.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DWWA15] Loris D'Antoni, Matthew Weavery, Alexander Weinert, and Rajeev Alur. Automata tutor and what we learned from building an online teaching tool. *Bulletin of the EATCS*, 117, 2015.

[Gor18] Vadim Goryainov. Pumping Lemma Game for Automata Tutor. B.Sc. Thesis, Technische Universität München, 2018.

[Hel17] Martin Helfrich. Kontextfreie Grammatiken in AutomataTutor. B.Sc. Thesis, Technische Universität München, 2017.

[HMU06] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Automata theory, languages, and computation. *International Edition*, 24(2.2), 2006.

[Kir13] Ross Kirsling. Directed graph editor. `http://bl.ocks.org/rkirsling/5001347`, 2013. [Online; accessed 19-February-2019].

[Lev66] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[Mai18] Sebastian Mair. Untestützung von Kellerautomaten in Automata Tutor. B.Sc. Thesis, Technische Universität München, 2018.

[Sch92] Uwe Schöning. *Theoretische Informatik-kurz gefasst*. Wissenschaftsverlag Mannheim, 1992.

[Wag17] Jan Wagener. Lehren von Algorithmen für reguläre Sprachen mithilfe von AutomataTutor. B.Sc. Thesis, Technische Universität München, 2017.