DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Webtool to create and check canonical LR automata

Leo Fahrbach

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Webtool to create and check canonical LR automata

Webtool zur Erstellung und Kontrolle von kanonischen LR-Automaten

Author:	Leo Fahrbach
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Michael Schwarz, M.Sc.
Submission Date:	15.02.2021

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

Munich, 15.02.2021 Leo Fahrbach

Abstract

This thesis presents the online learning tool *LR Tutor*. The tool is created to help students to understand canonical LR automata. The user can state a grammar and then create a canonical LR automaton. This automaton can be checked for correctness and the user gets appropriate feedback. This check can be disabled so assignments or exams can utilize this tool, without letting the students cheat. One version of the tool is hosted at TUM[1] and the code is published on github[2].

LR Tutor is based on the javascript library *mxGraph* to visualize and edit the automaton. The thesis' main part shows the changes and additions made to mxGraph to allow editing canonical LR automaton and not just graphs. An edit handler to edit the LR items of the state, and a connection handler to handle transitions between states, were implemented. Further the correctness check of the automaton is implemented.

The tool will be used next semester in the compiler construction course of TUM. There it will become clear, if the tool is a success and if the students can utilize its potential. In case a serious problem is found, it is very likely that the tool will be updated.

The thesis shows more feature for the future, however these may not be implemented ever. They are shown to add further potential to the tool.

[1]`https://www2.in.tum.de/lr-tutor`
[2]`https://github.com/leofah/lrtutor`

Contents

Abstract **iii**

1 Introduction **1**

2 Fundamentals **2**

3 Related Work **4**

4 Solution Approach **6**

4.1 Overview 6

4.2 Architectural Design 7

4.3 Components Of The Software 9

4.4 Design Problems 16

5 Future Work **18**

5.1 Features Related To Graph Construction 18

5.2 More Features For The Tool 21

6 Conclusion **24**

Bibliography **26**

1 Introduction

All students learn differently. For some students it is easier to learn with lecture notes, other students learn better by listening to lectures and yet other need to practice a lot to understand the contents of a lecture. Providing more learning options to students is a great way to improve the learning outcome on a topic. For many students creating canonical LR automata is not easy to understand, when they only have lecture notes but no way to train the creation. Especially the creation of canonical LR(1) automata is more complicated, due to the large size of the automaton and the handling of the lookahead sets. Drawing canonical LR automata on a paper is complicated, as in the beginning it is not known, how many states the automaton will have and how to layout the states without having transitions crossing too much. Even if a student has drawn the automaton on a paper they cannot easily check if their result is correct. One way is to look at a sample solution, in case one is provided. Another way is to compare the result with a fellow student. Students can profit a lot from a tool, where students can create the canonical LR automaton.

This thesis shows the tool *LR Tutor*[1], an online tool to build and check canonical LR automaton. The tool utilizes the javascript library mxGraph [8], which helps to create interactive graph applications. The user inputs a custom context free grammar, selects whether to create a canonical LR(0) or LR(1) automaton and starts to create the states and transitions of the automaton. While creating the automaton, the correctness of the automaton can always be checked, and the feedback can be applied immediately to correct the automaton in place.

Outline First, in chapter 2 the most relevant parts of canonical LR automaton are explained. Next, chapter 3 gives an overview on related parsing tools or automaton creation tools for students. In chapter 4 the design of LR Tutor are explained. It explains which requirements were needed and how they got implemented. Last, chapter 5 suggests different improvements on LR Tutor, which could be implemented in the future.

[1]`https://www2.in.tum.de/lr-tutor`

2 Fundamentals

This Chapter presents a brief overview on finite automata, context free grammars and canonical LR automata. The book from H.Seidl et al. [18] gives all the basics for compiler design, containing LR Parsers as well.

Finite Automata

Finite automata are used to recognize if specified regular language contains an input string [18, Chapter 2.2]. A finite automaton consists of a finite alphabet Σ, finitely many states Q, a start state $S \in Q$, final states $F \subseteq Q$ and the transitions δ. The transitions specify how to traverse the states of the automaton. They receive the current state and the current input symbol and give possible next states. For a deterministic finite automata (DFA) the there is exactly one transition for every state and letter. ($\delta : Q \times \Sigma \rightarrow Q$), whereas in a non-deterministic finite automata (NFA) there can be multiple transitions. ($\delta : Q \times \Sigma \rightarrow 2^Q$). The automaton accepts an input string, if there is a possible path in the automaton, which uses transitions with the next input character as their label, and ends up at a final state.

Context Free Grammars

Context free grammars are used to describe context free languages [18, Chapter 3.2.1]. A grammar consists of nonterminals N, terminals T (the alphabet Σ of the language), a finite set of productions P and a start symbol $S \in N$. A grammar is context free if the productions are arranged as follows: $P \subseteq \{A \rightarrow \alpha : A \in N, \alpha \in (N \cup T)^*\}$.

This example grammar is used in the thesis for shown automata.

$$\begin{aligned} S' &\rightarrow aAc \\ A &\rightarrow bbA|b \end{aligned}$$

All context free languages which are not regular cannot be parsed by a finite automaton, a more complex parser is needed. Parsing against a context free grammar can be realized with an LR parser, which reads the input from **L**eft to right and creates a **R**ightmost derivation. In comparison, an LL parser creates a leftmost derivation and cannot parse input strings for every deterministic context free grammar.

Canonical LR(k) Automaton

An LR(k) parser is based on a canonical LR(k) automaton, which is a special DFA, based on the grammar. As mentioned before DFAs cannot parse context free grammars. The canonical automaton is one part of the complete LR parser. The parser traverses the automaton with the input, but maintains a stack with all visited states of the automaton, therefore the travel path can be backtracked and context free languages detected. LR(k) Parser are *shift-reduce* parsers, they can either shift an input symbol onto the stack and traverse the corresponding transition in the automaton, or they can reduce a production and backtrack the production in the automaton. The parser can see at most the next k characters of the input, which decide whether the parser shifts or reduces.

An automaton is based on the context free grammar of the language. The alphabet Σ of the automaton consists of the terminals and nontermials: $\Sigma = T \cup N$. The states of the automaton are sets of LR(k)-Items. For the rest of the thesis the term *canonical LR automaton* is shortened to *automaton*, except when different automata (DFA, NFA) are mentioned and when it needs to be clear a canonical LR automaton is meant.

LR(k) Item An LR(k) item has the form $A \rightarrow \alpha_1 \bullet \alpha_2, a$ where $A \rightarrow \alpha_1\alpha_2 \in P$ needs to be a production and $a \in \{T^k\} \cup \{T^l\$: l < k\}$ is the current lookahead. The special lookahead character $ is used to denote the end of the input. For $k = 0$ the lookahead is omitted. For $k > 0$, $A \rightarrow \alpha_1 \bullet \alpha_2, \{a_1, \ldots, a_n\}(n \in \mathbb{N})$ is a short form for the set $\{A \rightarrow \alpha_1 \bullet \alpha_2, a_i : i \leq n\}$ of LR items.

An LR Item describes the current parse state and the expected lookahead: α_1 is already derived from the current input and lies on the top of the stack. α_2 needs to be found next in the input. If $\alpha_2 = \epsilon$ the complete RHS of a production $A \rightarrow \alpha_1\alpha_2$ lies 1on top of the stack and can be reduced by the parser.

It is proven, that LR(1) languages can express every deterministic context free grammar [18, Chapter 3.4]. When defining a grammar, a small k is wanted, so the lookahead sets and the number of states for the automaton are limited.

This thesis often mentions *LR(k) items*, thus they are just written as *items*.

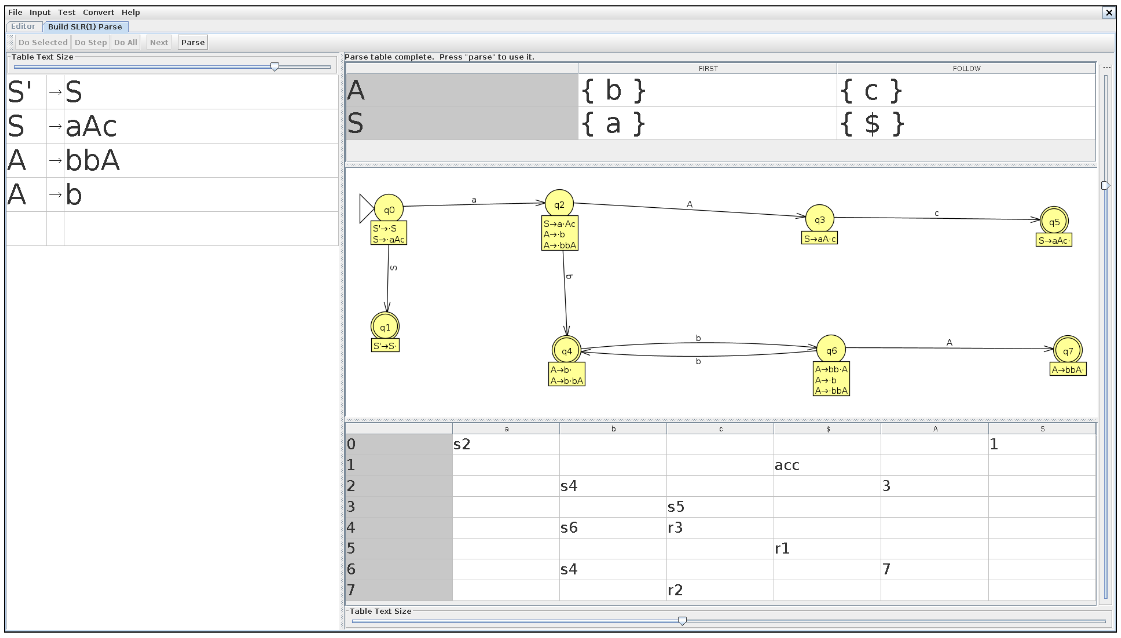
3 Related Work

Compiler construction courses often face the difficulty of teaching different parsing techniques. Hence, there are many tools which visualize the construction and execution of parsers, which are used at different universities. These tools can broadly be classified in two classes *Compiler visualizers* and *Parser generator visualizers*. *Compiler visualizers* are tools which show the process of parsing an input string either against a fixed grammar or against a custom grammar. *Parser generator visualizers* are tools which show how the parser is generated from the given grammar. The tools vary in the interaction with the user. The user can either create the parser by himself with guidance of the tool, or the tool shows directly the correct solution for the given input without any user input.

Compiler Visualizers show the user the compilation process step by step. The user can input a string to parse, and it is visualized how the syntax tree is generated. In PAG [16] the user can interact with the generated syntax tree and show the attributes of the nodes. The tools ICOMP [1] and VisiCLANG [13] are based on the fixed grammars PL/0, a pascal grammar, and CLANG respectively. Since both work on a fixed grammar, the parsing algorithm is optimized and both work as a handwritten recursive-descent parser. Other visualizers are based on compiler-compilers. The generated compiler parses the input and visualizes the parsing steps. Such visual compiler compilers are GYACC [10] which is based on YACC, and CUPV [9] which is based on the compiler-compiler CUP.

Parser Generator Visualizers show the creation of parsers. For LR Parsing the canonical LR automaton would be generated and shown. The first tools were LLParse and LRParse [3], which show the generation of LL(1) and SLR(1) parsers. Here the student can learn the differences between LL and LR parsing and why specific grammars are not LL(k) or LR(0) These tools are now combined with other automata tools to JFLAP [14, 11]. JFLAP can show the LL(1), SLR(1) and CYK parsers, and offers more tools for theoretical computer science, like DFA, NFA, and regular expression conversions. Other parser generator visualizers are PPVT [7] and PAVT [15]. Both tools receive a grammar, an input string and a selected parsing algorithm. The execution of the parsing algorithm is then printed to a file. or LR Parsing the output would contain the complete automaton and the syntax tree of the input. Jsmachines [2] is an online tool to generate parsers. The student can input the grammar and the input string to see the canonical LR automaton and the syntax tree of the input. Jsmachines provides the LALR(1), SLR(1), LL(1) and LR(1) algorithms. It can also simulate turing machines.

Conclusion The presented tools have helped students since parsing was improved with LR Parsers. They made the understanding of parser much easier for undergraduate computer science students. However, some tools are old and are not longer maintained, like VisiClang [13] or GYACC [10]. But old tools leave the space for newer tools like JFLAP [14], which is under constant development [11]. Figure 3.1 Shows JFLAP with a canonical SLR(1) automaton. LR Tutor, the tool presented in this thesis, is a parser generator visualizer where students can create canonical automata and are allowed to make mistakes. The created automaton is checked and feedback on the errors is given.



S'	→ S
S	→ aAc
A	→ bbA
A	→ b

Parse table complete. Press "parse" to use it.

	FIRST	FOLLOW
A	{ b }	{ c }
S	{ a }	{ $ }

	a	b	c	$	A	S
0	s2					1
1				acc		
2		s4			3	
3			s5			
4		s6	r3			
5				r1		
6		s4			7	
7			r2			

Figure 3.1: The SLR(1) automaton created by JFLAP, with first and follow sets and the parsing table

4 Solution Approach

This chapter will introduce the underlying architecture of the LR Tutor implementation. First we will give an overview of the main architectural design decisions. The process of choosing a fitting graph visualizer will also be shortly outlined. section 4.3 will then explain each component of the implementation in more detail. Lastly we will give a short summary of the implementation challenges faced while designing the tool.

4.1 Overview

The goal is, to design a tool, where students can create and check canonical LR automata. The idea was a similar setup as automatatutor [6, 5], where students can solve different graph problems and submit their solution for automatic correction. Automatatutor is used for homework assignments in the theoretical computer science course of the Technical University Munich. However, the effort to create a submission site, and an admin site where lectures can plan assignments and connecting students to lectures, is too much for one bachelor thesis. Only the correction of automata suffices for this thesis, no user or admin interface need to be implemented.

Following requirements need to be met by the tool. First the users can create canonical LR(0/1) automaton in a visual manner, so just a text representation is not enough. In a textual representation transitions between states cannot be detected easily. Next the user can check the correctness of their automaton and obtain feedback. Further the interface should be easy to use.

Similar to automatatutor LR Tutor can be be implemented as a web tool, since everyone can access and execute it, without considerable platform differences.

For these requirements the following components are implemented. First a visual and memory representation of the automaton needs to be developed. The editing and the checking of the automaton is based on this representation. Another component is the manipulation of the graph. This consists of three parts, first the editing of different states, second the transition between the states and last the selection of final and start states. Another required component is the correctness check of the automaton. Here the graph representation is checked for errors, which are then displayed to the user. A not required, yet useful component is the saving and loading of automata. It is easy to implement and helps students to continue with the problem later. All these components and core decisions are explained in detail in the rest of the chapter.

4.2 Architectural Design

Some designs for the tool have a huge impact on the rest of the tool. These architecture decisions are explained in this section. The main decision concerning the implementation is that between a client-server application, or a purely client based one. First it needs to be chosen, whether to use a client-server setup, or if a client setup suffices. Further the automaton needs a proper visualization, with the possibility to interact with the automaton, and not a visualization which shows one static picture of an automaton. The thesis considers these three javascript libraries: mxGraph [8], P5.js [12] and D3.js [4]. While P5 and D3 are only visualizations libraries, mxGraph is a library that supports graphs natively.

Application Architecture

There are two main ways to implement the tool. Either a client-server setup, where the server can check the submitted automaton and save automaton for later use, or solely a client setup, where the automaton check is done on the client. Then only a webserver is needed to serve the webpage. This section presents the positive and negative aspects for every architecture.

A Client-Server architecture allows checking the automaton on the server side. This is mainly useful for homework submission, when students should not be able to cheat. The server can control how often one student submits an automaton, and the correction result cannot be compromised. Further the server can give some useful features like saving created automata and later load them from the server. This enables shareable URLs where students can look at an automaton without creating it themselves.

The drawback of a server backend is mainly that there are two separate programs to implement, first the frontend at the client, and the backend at the server. Further the communication between these two programs need to be clear and without any problems. This makes the tool much more complicated and prone to errors.

A Client Only architecture moves all the functionality to the web frontend. The main advantage is the easier set up, as only a static webpage needs to be served. Further the website can be used offline, a student can load or download the site in advance and is still able to create and check automata, even if he is not connected to the internet.

With a client architecture the students can share their solutions with no notification to the server, as the check is done on the client side, without server notification. This makes it not appropriate for homework correction, as a student can improve the automaton very often and share the correct result with fellow students. As exactly one automaton is correct, it is difficult to determine whether a result was really from the right student.

With the easier setup of a client only architecture this architecture was chosen for LR Tutor. It can handle every requirement for the tool. With the server architecture set it can now be decided what graph visualization library to use.

Choosing The Visualizer

The other large design decision is the visualizer to used to display the automaton. Somehow the created automaton must be shown on the webpage. This can be accomplished for example with SVG (Scalable Vector Graphics). However, writing plain SVG without any support is really difficult and may not work on every browser. To show an automaton, a representation of a graph should be displayed at any time. There are many libraries, that support different visualizations on web pages. Popular libraries are *d3* [4] and *p5* [12]. The library *mxGraph* [8] is specialized to show graph like diagrams. This chapter compares these different libraries and states why mxGraph is chosen.

P5 is focused on artists, designers, educators and anyone else [12]. P5 uses the HTML canvas element to draw shapes on the screen. A P5 program has two main functions, a setup routine and a draw routine. The setup is called once to initialize the canvas and all needed variables, the draw routine is called on every frame (default 60 fps) to update the contents of the HTML canvas element. The focus is on creating animations, like animated algorithms or nice design pictures, rather than displaying a graph, which does not change every frame, except when a movement of a state, or a similar action, is visualized. As the focus is on design, not on interaction, P5 is not suitable for the development of LR Tutor.

D3 is short for **D**ata **D**riven **D**ocuments [4]. It is used to change the Document Object Model (DOM) based on data. One part are the introduced shortcuts to handle DOM operations on a larger scale as with the default DOM API. This is similar but different to jQuery[1]. Another part is to visualize arrays of data in any form, e.g. a statistic data as a bar chart or a pie chart. The data is not fixed and can change over time, however the visualization is not bound to static data, the data can be updated and the visualization

[1] `https://jquery.com/`

updates too. In the case for LR Tutor the data would be the representation of the graph and the visualization would be in SVG. This is the approach used in automatatutor [6] to visualize finite automata.

Using D3 implies to maintain and update an SVG image by hand, which is difficult, as SVG is a really complex language. Automatatutor yet uses D3, however finite automata are less complicated than canonical LR automata. In a finite automaton the states have a numbers as text and therefore do not change in size or have editable states, whereas a canonical LR automaton has many LR items as text for states. Despite D3 being used in automatatutor and having a lot of capabilities to manage SVG, it would still be too complicated to visualize canonical LR automaton with D3 and is therefore not used for this tool.

mxGraph is a library which is focused on creating, editing and visualizing graphs [8]. It has support for many browsers and hides the SVG interaction to the programmer. Actually it can use different visualization methods, like HTML canvas or SVG, based on the configuration or the browser. MxGraph represents the graph with different classes and has a lot of functionality built in, like handling movement of states, connecting states, selecting states, editing states, grouping of states and many more features. Very similar features are needed for an automata creation tool.

As an automaton is a graph, this library will be used to visualize the graph and handle editing the graph. Unfortunately mxGraph got deprecated on 09.11.2020[2], shortly after registering the thesis. The implementation was already based on mxGraph, therefore chancing to another library and writing everything from scratch was not feasible. Javascript is such a widely used language, hence mxGraph is likely to work for many more years, as adding breaking changes to a widely used programming language is not reasonable. However, if there is a critical bug in mxGraph, it will not be patched, further the support and documentation will not improve, and the library will soon be removed from npm (node package manager). This website does not use the npm version, so the removal will have no effect on this tool. Jgraph is searching for a new maintainer, maybe one is found soon and it is fully supported again. Concluding, mxGraph is the best of the analyzed libraries to visualize a graph in a webpage.

4.3 Components Of The Software

LR Tutor consists of different components, which interact with each other to enable the creation and correction of automata. As mxGraph is the base of the complete tool, a short introduction on the usage and integration of mxGraph is given. The user

[2] `https://github.com/jgraph/mxgraph`

needs to state the problem he wants to solve, hence he writes a grammar and selects whether an LR(0) or LR(1) automaton is build. Next the automaton is created by the user. MxGraph already provides a lot of edit functionality, however some functions needed to be changed to fit the creation of automata. This is achieved with two parts, one for editing the states and their LR items and the other to add transitions between states. After the automaton is created, the tool checks, whether to user created the correct automaton. The tool checks several elements of the automaton and errors are shown to the user, either directly in visualization or as text next to the canvas. While creating the automaton the user may want to have backups or save the result. LR Tutor provides a save and load feature, to back up the current automaton. This feature is based on the encoding of an mxGraph to XML.

MxGraph Usage

MxGraph provides a lot of functionality to handle a graph. The main three classes are `mxGraph`, `mxGraphModel` and `mxCell`. The states, transition and every other element on the canvas is an `mxCell`. The `mxGraphModel` holds the layout and elements (mxCells) of the graph. `mxGraph` is the class containing everything associated with the graph, it handles the access to the model and fires many events in case the graph is updated.

The events are fired at specific actions and run the complete handling in mxGraph. There are events for editing, moving, adding, removing, resizing cells and many other events. When utilizing mxGraph these events are the key to create and edit a graph. Especially as every user input is transformed into a corresponding event.

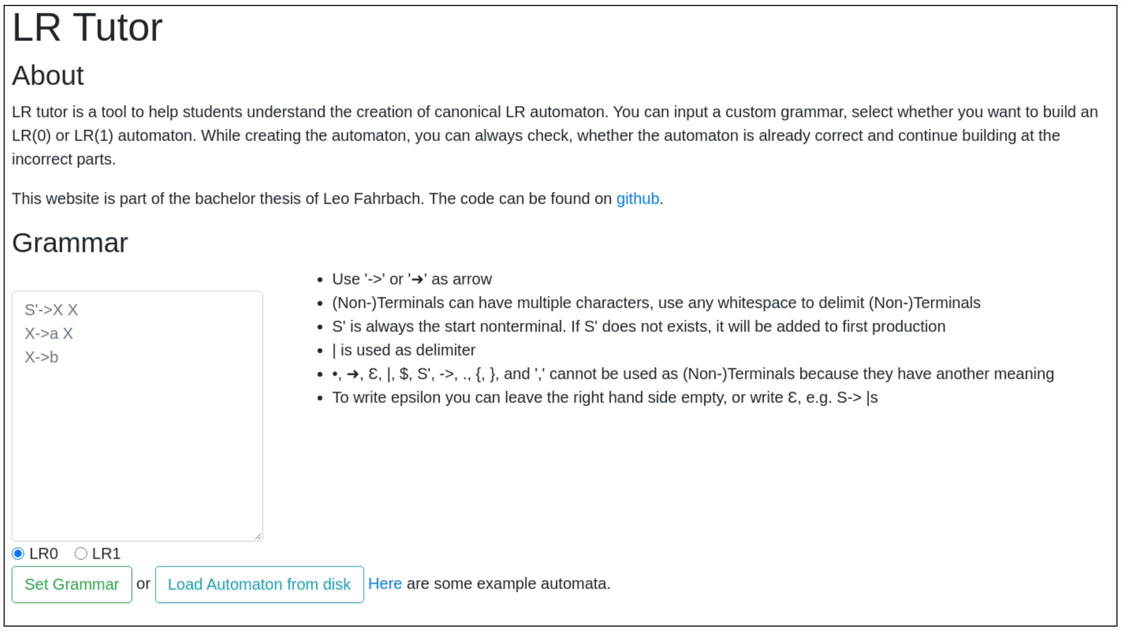
MxGraph already has many event listeners handling e.g. moving states, or connecting states. These listeners are automatically added when initializing `mxGraph`. As many features are implemented differently or are completely obsolete, these need to be deactivated when initializing the graph, like the cloning or resizing of cells. Other features do not have the correct effects to create an automaton and therefore need to be adapted. Sometimes it is difficult to understand what features mxGraph provide and how to disable specific features.

With this many features already implemented in mxGraph, the focus on the implementation lies on enhancing the features to be suitable for automata.

Grammar Handling

There are many ways to input a context free grammar. For this tool the input should be easy to read, write and should not be restrictive. Different right-hand sides (RHS) of the same nonterminal can be delimited by `'|'`. (Non-)terminals can have multiple characters in their name, this allows a more expressive naming. This was included, as the example grammar from the compiler construction lecture at TUM has the terminal *int*. To delimit (non-)terminals on the RHS whitespace is used. S′ is always used as start nonterminal, if it is not included in the grammar, the first production will be augmented to add S′ as start symbol. Keyboards do not have special characters like ϵ or $\rightarrow$, they are inserted as follows: For $\rightarrow$, -> can be written. Other context free grammar descriptions use '$\sim$' or 'λ' as epsilon. However, for LR Tutor the RHS can be left empty to add an epsilon production. This implies, epsilon inside the RHS (A$\rightarrow$aϵb), are not supported, but such productions are not needed at all. Some special characters ($, {, . . .) can not be used as a (non-)terminal, however, it is obvious, why they are not allowed. Figure 4.1 shows the start of the tool and a grammar can be inserted.

The grammar class in the tool has more functionality than parsing the input grammar. It handles all productions, the LR items can be parsed as well. With the parsed items, their closure or shifted item can be computed, which is needed for the correctness check of the automaton.


LR Tutor

About

LR tutor is a tool to help students understand the creation of canonical LR automaton. You can input a custom grammar, select whether you want to build an LR(0) or LR(1) automaton. While creating the automaton, you can always check, whether the automaton is already correct and continue building at the incorrect parts.

This website is part of the bachelor thesis of Leo Fahrbach. The code can be found on github.

Grammar

```
S'->X X
X->a X
X->b
```

- Use '->' or '➔' as arrow
- (Non-)Terminals can have multiple characters, use any whitespace to delimit (Non-)Terminals
- S' is always the start nonterminal. If S' does not exists, it will be added to first production
- | is used as delimiter
- •, ➔, Ɛ, |, $, S', ->, ., {, }, and ',' cannot be used as (Non-)Terminals because they have another meaning
- To write epsilon you can leave the right hand side empty, or write Ɛ, e.g. S-> |s

[✓ LR0] LR1

Here are some example automata.

Figure 4.1: New Loaded LR Tutor, where the user can input a grammar or load a saved automaton.

Automaton Creation

This section explains the editing and creation of the automaton. The basic handling is already implemented in mxGraph, however some features are adapted to create an automaton. A new `editHandler` and `connectionHandler` were implemented. The edit handler, handles the adding, removing and editing of LR items in states and the resizing of states. The connection handler handles the adding of transitions between states. A start state and (non-)terminal is selected to add the transition. Then the ending state of the transition is selected and the transition is added to the automaton.

Edit Handler The objectives of the edit handler are to transform an inserted LR item and to dynamically adapt the number of LR items in a state. The edit handler listens for `EDITING_STARTED`, `EDITING_STOPPED` and `DOUBLE_CLICK`. The editing events are fired by the mxGraph edit handler, it denotes the start and end of editing a cell text. Actually only *editing stopped* is needed, however *editing stopped* has no information, which cell was edited. However, the editing started event contains the edited cell. This cell needs to be remembered until editing stopped is fired.

When the user finished editing an LR item, the text is transformed to set '->' to '$\rightarrow$' and '.' to '•'. If the edited item was the last one in the state, a new item will be added to the state, which can be edited now. Hence, the user only needs start editing a state once and can then input all LR items for the selected state, without touching the mouse in between editing two items. The double click event is used to directly start editing a new LR item at the end of the clicked state. The mxGraph edit handler uses the double click too to start editing the clicked cell if it has text. If the user is finished with editing a state, the state is resized to a proper size. To summarize, editing a state is either done by double clicking the item, which needs to be edited, directly or by double clicking the state to add a new item to the end of the state.

Connection Handler In mxGraph there is already a connection handler. This handler can add edges between any vertices of the graph. However, it does not support edges with labels, which are needed for an automaton. Further, the automaton needs to be deterministic, hence there needs to be a method to prevent adding the same edge to a state twice. The built-in connection handler is disabled, and a new one for deterministic automata is implemented. Figure 4.2 shows the new active connection handler.

When exactly one state is selected, buttons are shown to select the terminal to use for a transition. After selecting the (non-)terminal a preview of the new transition is shown on the canvas, while moving the mouse to different positions. If the transitions ends on a blank spot on the canvas, a new state will be added at this position, and the user can directly edit the items of the new state. Otherwise, if the transition ends on another

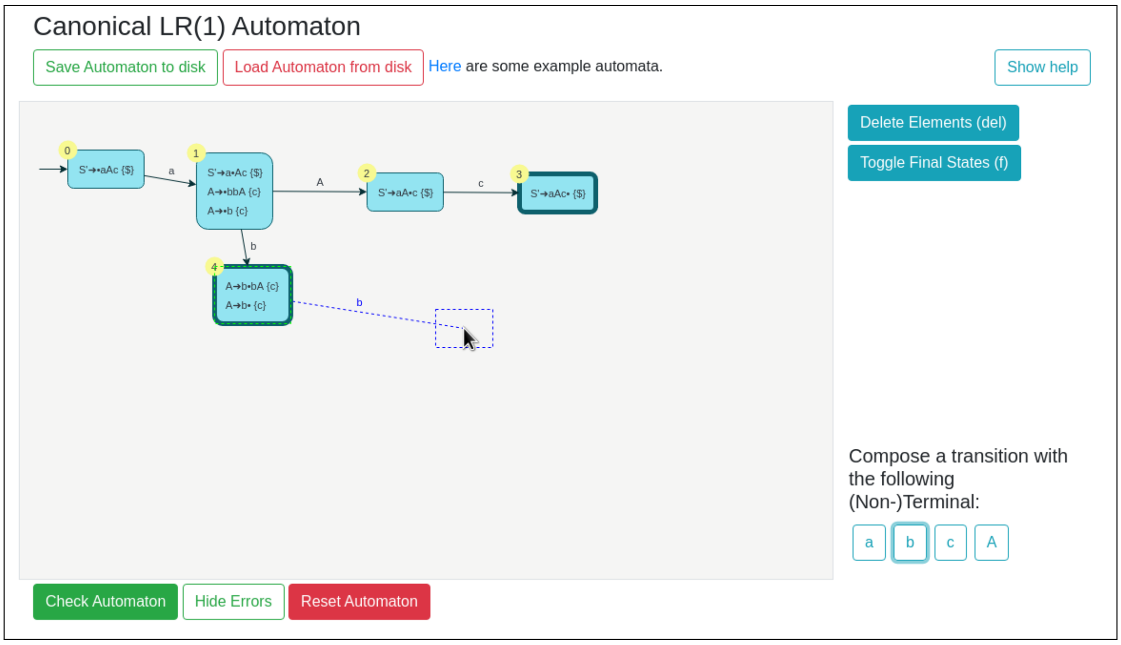


Figure 4.2: The user is editing the automaton and adding a new transition

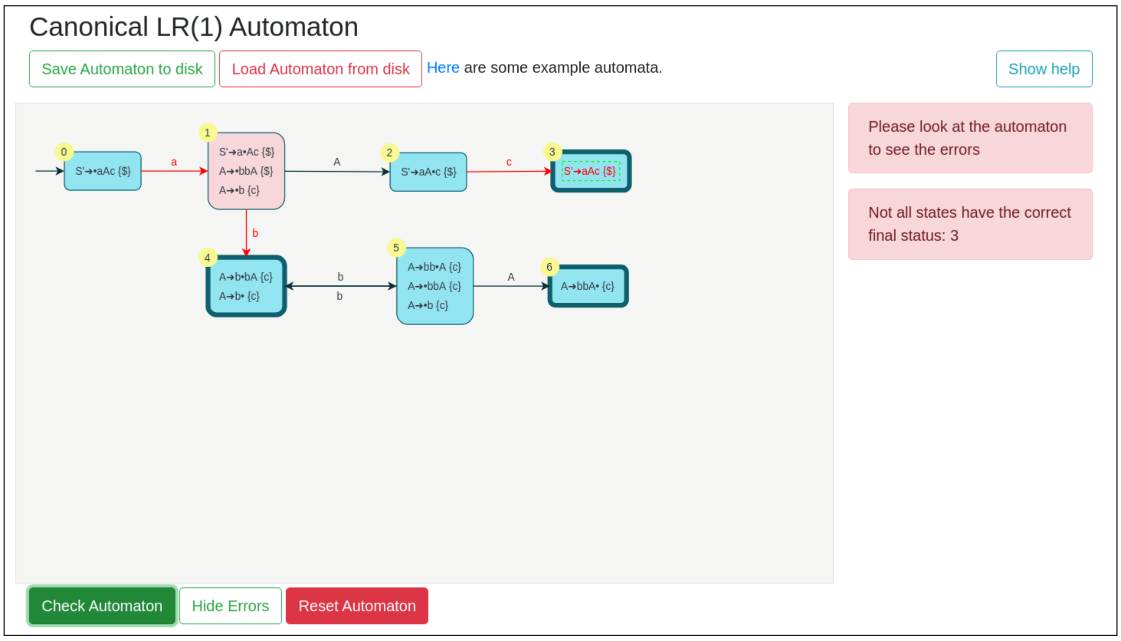
state, this state is selected as the ending state for the transition. In case a transition with the selected (non-)terminal already exists at the start state, the old transition is removed and the new one stays, the automaton is always deterministic.

With the edit handler and the connection handler implemented, the automaton can be created by the user. Now the correctness check of the automaton needs to be implemented.

Checking The Correctness

The next important component is the correctness check of the automaton. Students should be able to receive feedback on their automaton to see the errors they made. The check is done on basis of the user's automaton not in comparison to the correct internally created autoamaton. Only the integrity of each state and the transitions between the given states need to be checked. The different checks can be classified in three classes, input checks, automaton checks and advanced graphical feedback.

Input Checks show invalid or too broad input to the user. It shows LR items which cannot be parsed correctly and thus are ignored for every other check. Further, duplicate states or LR Items are shown, as they are not needed and increase the size of the automaton unnecessary. At last, not connected states are shown. All these checks ensure that the created automaton is minimal.



Here are some example automata.

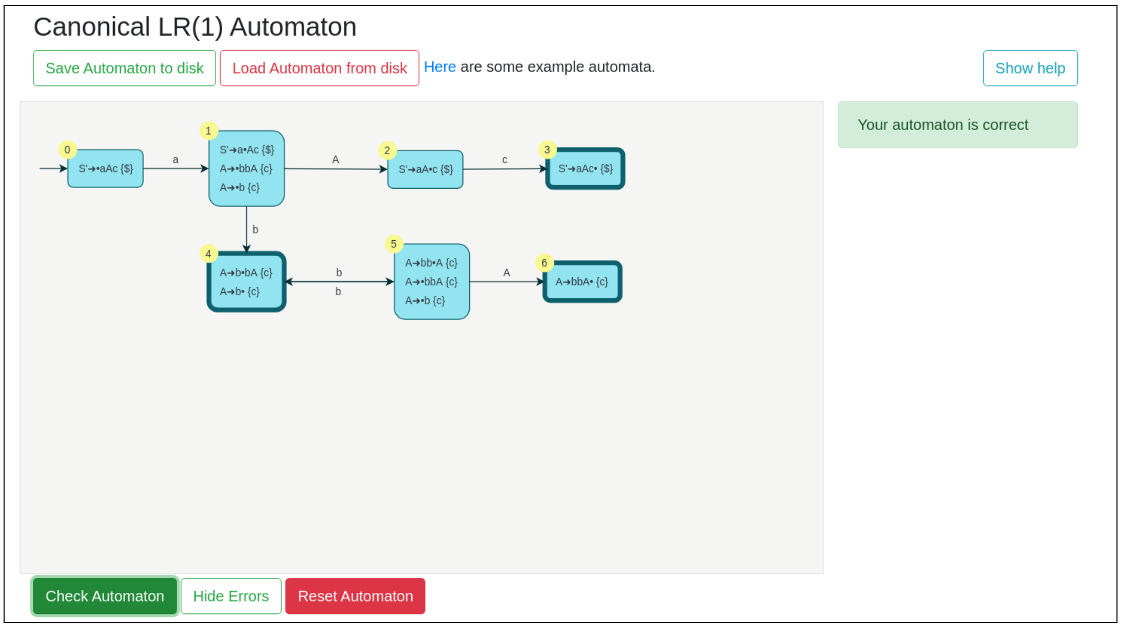
Please look at the automaton to see the errors

Not all states have the correct final status: 3

Figure 4.3: The automaton has some mistakes, which are shown on the canvas and as text next to the canvas

Automaton Checks test the definition of the automaton. First, the *transition check* tests, if every state has the correct transitions it needs. For example if a state has only the item $X \rightarrow \bullet aA$, only the transition with a is allowed, every other transition is incorrect. Further, the transition is only correct if the target of the transition has the correct closure for the shifted LR items. In this case the target state needs to be $closure(X \rightarrow a \bullet A)$. Additionally, the start state is checked separately, as it has no incoming transition which would assert the closure during the transition check. The *start state check* only checks if the start state has the form $closure(S' \rightarrow \bullet X)$, The outgoing transitions are already checked in the transition check. Further, the final states are checked as well. Every state containing an LR item $X \rightarrow \alpha\bullet$ needs to be marked final. These three checks test the complete automaton against the definition of canonical LR automata.

Graphical Feedback One extra check is included to show incorrect closures of states directly on the canvas. Of course the correctness of the closures is already tested in the transition check, however these states are not marked on the canvas yet. Hence, the



Here are some example automata.

Your automaton is correct

Figure 4.4: The automaton is now correct and the check has no error found

check tests $lrItems = closure(lrItems)$ for the items of a state instead of the closure of shifted items. This lets the user identify if the closure is in itself correct and if an errors lies in the closure or in the transition.

Figure 4.4 and 4.3 show nearly the same automaton and their feedback after the correctness check. All these checks help the user to create the automaton, and he can learn what his mistakes were.

Saving And Loading The Automaton

The last component of the tool, is the saving and loading of automata. It is based on `mxCodec`, the encoder from mxGraph, which can serialize and deserialize an mxGraph-Model as XML. However, the graph alone is not every information which needs to be saved. The grammar, the LR(0) or LR(1) option and the start state information need to be saved too. The grammar is stored in plaintext and parsed again, if the file is loaded. These values are added as separate XML attributes and elements to the file.

For compatibility reasons a version number is added as well. The tool is designed to only write and read only one specific versions. Hence, if the saving mechanism is changed to store more aspects, the version number can be adapted accordingly. The current version supports all needed features, so file version 1 is likely to stay.

As the model can contain temporary cells, like the preview for a transition, they need to be removed before saving the automaton. If they were saved too, they would stay on the canvas all the time and never be removed on the loaded graph. Data which can be removed automatically, like error highlights or the state ids, can be saved. They are updated accordingly after a file was loaded.

Saving and loading is not only writing the graph model to a file, more metadata of the problem and the graph need to be saved.

4.4 Design Problems

This last section presents a short summary of some problems which were encountered while implementing the tool. The largest issue was the integration with mxGraph. MxGraph has many functionalities, but they are only sparsely documented.

At one test we noticed that cells can be cloned when the holding *ctrl*. It is nice that mxGraph supports cloning, however this leads to a lot of problems with automata, as the transitions and states can be messed up after they were cloned. Somewhere needs to be an option to disable the cloning, however the documentation has no explanation which variables determine the cloning behaviour. It turns out that there are different ways to disable cloning, either by disabling it completely in the mxGraph or by disabling it only for specific cells with a style attribute. However, in the documentation is no reference to the other variable. Cloning could be enabled for specific cells again, after a deeper check of the cloning is done. Often states of an automaton are similar, especially for LR(1) automaton, hence cloning can be an option enabled again.

Another problem was managing the final status of a state. Either the status is stored in a global list, which contains every final state, or the status is stored directly in the mxCell. For both options there needs to be a method to serialize the final status of cells without large overhead. The solution was to store the type and final status information directly in the mxCell. For this the existing variable `style` of the mxCell is used, as the appearance of a mxCell is directly connected to the status the cell has. Further, the style is automatically serialized by mxGraph and no further code is needed to save the status. However, the style could contain additionally style tags, e.g. to show a red color for errors.

A lot of time was spent, figuring out, what function does exactly what was intended to do. Overall mxGraph is missing a detailed documentation. A documentation enables an easier access to the library. However, it is understandable, that maintainers may not have the focus on a perfect documentation, especially if the library is not so popular. After many hours with mxGraph, all the functionalities come slowly together and the handling and options become clearer.

5 Future Work

The implementation of learning tools is never fully finished. There are many features that can be implemented to improve the learning experience for students. The tool is developed for the basic capability to create and check canonical automata, but some features are still missing. Not every possible feature is mentioned, as the tool can be extended in a larger scale. Maybe after an extensive use of the tool more obvious aspects will be found by students, which were not found while testing the tool beforehand.

The features can be categorized in two main parts. The first category focuses on features that directly affects how the user can create the automaton. For example, the functionality to show or create the correct automaton immediately affects how the user will create their solution. The other category is for actions apart from creating the canonical automaton. It is about changing the focus of the tool to not only create and check the canonical automaton but add more learning capabilities. This includes the possibility to build the parsing table from the automaton, or a tutorial on canonical LR automaton.

5.1 Features Related To Graph Construction

These features are focused on direct changes of the capability how the student creates the canonical LR automaton. Some changes, which are not a new feature, but problems in the current implementation are mentioned as well.

Small Improvements For The Current Version

All in all, handling of user input is complete with many options to edit the automaton. However, there are still some inconsistencies when creating the automaton. To fully fix these inconsistencies more research into the inner works of mxGraph is needed. On these actions it is nice to have more insight on how to patch them best.

One inconsistency happens while adding a new state to the canvas, when the canvas is clicked. MxGraph has an event for `DOUBLE_CLICK` but for each of the clicks a `CLICK` event is fired as well. When double clicking the canvas two new states are added, which is not intuitive. Indeed, there are many more cases, when a state is created on the canvas, but it was not the intention of the user to create a new state.

Another part is the handling of `EDITING_STOPPED`. If the user finished editing an item, `EDITING_STOPPED` will be fired. If this was the last item in the state, a new item will be generated in the state. However, if the current item is finished it may not be the intention of the user to further edit the current state, for example if another state is selected, and the cause of the editing stop was the click on the other state. MxGraph has no option to determine why exactly the editing is stopped now. The tool needs an algorithm to determine what caused the stop and how to proceed with creating new items. Another solution would be to overhaul the editing handling with a completely different approach.

Build And Show The Correct Automaton

Currently, only the user creates the automaton by adding each item, state and transition by himself. For larger grammars this is a long task, and many mistakes can happen. Of course the point of this tool is to learn from these mistakes, however sometimes users prefers to see the correct automaton for a given grammar, like JSMACHINES [2] does this for LR(1)-Automaton. Generating the correct result is one way for supervisors of a lecture to find appropriate problems and compute the result without any effort.

It is not hard to implement as the correctness of the graph is already checked. For this check the closure of each state is computed and also the items are shifted to obtain the next item after a transition. These given functions just need to be applied accordingly to build the correct graph. Most of the features to build the automaton are very similar to the one checking the correctness of the automaton.

Creating the correct automaton leaves the question when and where to allow the creation of the automaton. There are different approaches to integrate this feature. Initially, the creation of the automaton can be a completely new tool with no interaction, only the correct automaton is shown. This tool would create the automaton for reference. Next it can be integrated on the learning tool, add a button to let the computer create the automaton. The created automaton can then be modified with the actions allowed by the tool. So the automaton can be saved and loaded again and be edited. If this feature is added, supervisors of homework or exams need to be sure, whether and how the correct automaton creation can be abused by the students, such that they will not copy the solution.

All in all, it is great to have a tool which shows the correct automaton. However, it needs to be decided how to integrate this feature into the tool, such that students cannot cheat but still it is a help for students and supervisors.

Undo Action

To create the automaton many actions can be used, like adding a state, editing the items, adding transitions, deleting states and transitions and many more. If the user makes a mistake, like deleting a very large state with many transitions and items, all the progress is lost and can only be restored, if the user does the tedious work of adding the state and the transitions again or if the automaton was saved to the disk beforehand.

It would be really nice to have an *undo* feature, to undo the last actions. To create an undo-system every action needs to be classified, whether this action needs to be undoable. Actions like mouse movement or selecting a terminal for a transition do not need to be undoable directly. Only actions, like adding a transition to the automaton, needs to be undoable. Further it needs to be clear how to undo the action to keep the graph in a consistent state. There already exists an `mxUndoManager` in the mxGraph Library. However, a programmer needs to figure out how this manager works and how easy it is to integrate it in the current application. The mxUndoManager is a starting point to create an undo-system for the graph.

Multiuser Support

Another improvement for the tool is a multiuser capability, like in *etherpad* [17] or similar applications. This means multiple users on different machines can edit the same automaton at the same time, cooperating to find a solution.

There are many components to design. First, a new server backend is needed to communicate with the clients, as the current setup is a static website which serves the files, but cannot handle any communication of clients. Further, there needs to be a failover if a user looses connection, and the automaton has an inconsistent state. There needs to be a way to determine the correct automaton and synchronize the clients again. Next, there can be many race conditions between the clients. If two users update the automaton at the same time, an algorithm is needed, such that both updates are incorporated and no update is lost due to a race condition. Last, if there is an undo-system this undo-system also needs to be synchronized between all clients such that they all undo the same actions.

A very basic multiuser support can be implemented easily. There exists a way to save the automaton to the disk. This feature can be expanded to send the saved automaton to the server and update all the clients with the new automaton. However, this has a high potential for race conditions, and the users cannot share the same undo-system.

In `mxGraph` there is no implemented capability to synchronize different users of the graph, but there is the `mxEvent CHANGE` which fires after each change of the graph. With a listener, which receives and compiles the actual change, and a corresponding backend, which propagates the change, multiuser support can be built with less effort. However, multiuser support still is a really difficult task as the core architecture of the tool needs to be changed and there can be many race conditions. A very basic multiuser support, based on saving and loading the graph by hand, can be implemented easily.

5.2 More Features For The Tool

Apart from improving the creation of the automaton, other features can be added which increase the learning options for the tool. The tool can contain a section to create and check the parsing table of an automaton, or add a tutorial on canonical LR-automaton. Further a complete learning platform can be built to pose and correct problems.

Creating And Checking The Parsing Table

A canonical LR automaton is not yet a shift reduce parser. A shift reduce parser has a parsing table to determine which action to do next. The action can either be to *shift* an input token to the stack, *reduce* the top of the stack according to one production, *accept* or *reject* the input. The information which action needs to be applied next, is given in the parsing table. The purpose of the automaton is to create the parsing table and so a parser for the grammar.

There are different methods to generate the parsing table from the canonical LR(0/1)-Automaton. Most notably LR(0), SLR(1) and LR(1). The tool can have a utility to create a parsing table for a given automaton. The created table is checked by the computer, and the student receives appropriate feedback. Creation of the automaton and the parsing table can be combined to one workflow, so that students learn the complete process of generating an LR parser.

Tutorial On Canonical LR Automata

The tool's purpose is to help students understand the creation of canonical LR automata. However, there is no information given how such automata are built and why they work, the students only create an automaton, and the tool shows the wrong parts in the automaton. The lecture teaches the information how automata work and how they are built. To supplement this tool an explanation about automata can be added.

The explanation can either be broad and just cover the relevant parts needed to build the automaton, or it can be deep and explain why the generated automaton works. The Lecture explains the theory of the automata thus a broad explanation would suffice for the average student. The broad tutorial can consist of the main mathematical definitions for the closure, the transitions, the start and end states. For LR1 automata a definition of *first* and *follow* sets is needed as well. For each definition in LR(0) and LR(1) an example can increase the perception, especially for edge cases. A reference of the important definitions makes for an easy, yet powerful explanation.

Grading And Homework Integration

Another idea for the tool is homework integration. This means lecturers can add their students to a homework group and submit different problems for the students to solve. The students use the tool to solve the problem and submit their result to the server to be graded. The tool can already check the correctness of automaton, hence the check algorithm can be tweaked to implement a grading algorithm for the students. Automata Tutor [6, 5] already hosts such a homework tool where different basic problems of theoretical computer science can be solved, but it has no option for canonical LR automata.

To implement this feature a huge architecture change is needed. A login and registration for students and teachers, a frontend to pose and solve problems, a backend to grade the solution, and many more components are needed. Further the tool already gives feedback on the automaton. It needs to be decided if this feedback is removed, so that it cannot be abused, and all the grading is merely performed at the backend. As well, it needs to be decided whether the save and load feature of graphs stays enabled, as it gives an easy form to pass the correct solution among students, and they do not learn anything and just copy the homework.

The idea of a connected learning platform has potential, however for the correction of LR automata, the outcome for the implementation effort is limited. Here are multiple reasons why building automata is not useful for graded student homework. Creating an automaton more like a recipe, as only the definitions of the lectures are applied. Checking the solution will only show if the student understands the definitions of canonical LR automata, not if the student knows how LR parsing works. The problems presented in Automata Tutor are problems where the students need to think and be creative to find a solution instead of just applying rules. Next, the creation of an LR automaton is tedious, especially if the grammar is large and very similar items need to be written many times. The process is *not* like solving a puzzle, it rather checks the concentration of the student, whether the student wrote every item correct and forgot no transitions. Additionally, the difficulties of creating different automata are nearly the same, whereas in Automata Tutor easy and hard problems can be posed. Usually the automaton and the finished LR parser is generated by an LR-Parser-Generator. For large grammars people cannot create the parser by hand. In contrast, the problems given by Automata Tutor are mostly not solvable by a computer. Integration in homework is dispensable for the given reasons.

The grading of automata should be focused on the definition rather than on the details, as there can happen many careless mistakes when writing many items. However, checking the concepts without the details is hard for computers to achieve. All in all grading helps the lecture's correction time, but it is difficult to create the architecture for a submission and automatic correction site. Further, automaton creation may not be used often as homework in a compiler construction course. LR Parsers are only one small part of the course.

6 Conclusion

In this thesis introduced the tool LR tutor. The tool helps students to understand the creation of canonical LR automata. They can create a custom automaton and the tool shows the errors of the automaton to the student.

First we summarized the fundamentals of finite automata, context free grammars and the basics of canonical LR(k) automata. Finite automata are used to parse strings against a regular language, an LR parser parses, with the help of a canonical LR automaton, strings against a context free language. The states of LR automata are based on LR items.

Next we had a look on related learning tools. Universities implemented a lot of different tools, to help students to understand different parsing algorithms. Some tools only show the parsing process step by step, and other tools show the complete creation of the parser. JFLAP [14, 11] is the most advanced tool and even provides other features than only parsing algorithms. In contrast, LR Tutor is only focused on LR(0) and LR(1) parser creation.

In the next chapter we presented the design of LR Tutor. The application architecture and the visualization library are chosen appropriately. A frontend based tool suffices for the creation and check of automaton. The javascript library *mxGraph* is used to show and edit the automaton on the webpage. It has native support for graph structures. In the frontend the user can input a context free grammar and build the automaton based on this grammar. MxGraph provides a lot of editing features to the automaton. The editing of LR items and the connection of states needed to be changed, such that canonical LR automaton are created. However, using a large library has some drawbacks. Some special functionalities needed to be disabled without exactly knowing how to accomplish the wanted results.

In the end we glanced at future features of LR Tutor. Many features can be added to improve the learning experience of the students. One feature is the adding of a tutorial on LR Parsers, another feature is enabling multiuser support over the server.

LR Tutor can help many students in next semesters to understand the concept of LR parsing. The tool will be tested in the next semester (summer 2021) at the compiler construction course of TUM. Then it will become clear, if the tool is accepted by the students and reaches the desired goal. The code is found on github[1] and can be hosted

[1] `https://github.com/leofah/lrtutor`

anywhere. TUM hosts one version here[2].

All in all it was a great experience to implement a learning tool and hopefully it will help many students. Yet another tool to learn how to parse is finished.

[2]`https://www2.in.tum.de/lr-tutor`

Bibliography

[1] K. Andrews, R. R. Henry, and W. K. Yamamoto. “Design and Implementation of the UW Illustrated Compiler.” In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 105–114. ISBN: 0897912691. DOI: `10.1145/53990.54001`.

[2] G. Apou. *LR(1) Parser Generator*. `http://jsmachines.sourceforge.net/machines/lr1.html`. Accessed January 14, 2021. 2011.

[3] S. A. Blythe, M. C. James, and S. H. Rodger. “LLparse and LRparse: Visual and Interactive Tools for Parsing.” In: *ACM SIGCSE Bulletin* 26.1 (Mar. 1994), pp. 208–212. ISSN: 0097-8418. DOI: `10.1145/191033.191121`.

[4] M. Bostock. *Data-Driven Documents (D3)*. `https://d3js.org/`. Accessed February 08, 2021. 2021.

[5] L. D’Antoni, M. Helfrich, J. Kretinsky, E. Ramneantu, and M. Weininger. “Automata Tutor v3.” In: *Computer Aided Verification*. Ed. by S. K. Lahiri and C. Wang. Cham: Springer International Publishing, 2020, pp. 3–14. ISBN: 978-3-030-53291-8.

[6] L. D’Antoni, M. Weaver, A. Weinert, and R. Alur. “Automata Tutor and what we learned from building an online teaching tool.” In: *Bulletin of the European Association for Computer Science* 117 (Oct. 2015), pp. 143–160.

[7] A. Jain, A. Goyal, and P. Chakraborty. “PPVT: A Tool to Visualize Predictive Parsing.” In: *ACM Inroads* 8.1 (Feb. 2017), pp. 43–47. ISSN: 2153-2184. DOI: `10.1145/3002136`.

[8] JGraph Ltd. *MxGraph Website*. `https://jgraph.github.io/mxgraph/`. Accessed January 28, 2021. 2020.

[9] A. Kaplan and D. Shoup. “CUPV—a visualization tool for generated parsers.” In: *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’00. Austin, Texas, USA: Association for Computing Machinery, 2000, pp. 11–15. ISBN: 1581132131. DOI: `10.1145/330908.331801`.

[10] M. E. Lovato and M. F. Kleyn. “Parser Visualizations for Developing Grammars with Yacc.” In: *Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’95. Nashville, Tennessee, USA: Association for Computing Machinery, 1995, pp. 345–349. ISBN: 089791693X. DOI: `10.1145/199688.199855`.

[11] I. C. McMahon. *Improving the Capabilities of JFLAP*. Undergraduate thesis, Duke University. 2014.

[12] Processing Foundation. *P5js Website*. `https://p5js.org/`. Accessed February 08, 2021. 2021.

[13] D. Resler. “VisiCLANG—a Visible Compiler for CLANG.” In: *SIGPLAN Not.* 25.8 (Aug. 1990), pp. 120–123. ISSN: 0362-1340. DOI: `10.1145/87416.87483`.

[14] S. H. Rodger and T. W. Finley. *JFLAP: An interactive formal languages and automata package*. Jones and Bartlett, 2006. ISBN: 0763738344.

[15] S. Sangal, S. Kataria, T. Tyagi, N. Gupta, Y. Kirtani, S. Agrawal, and P. Chakraborty. “PAVT: a tool to visualize and teach parsing algorithms.” In: *Education and Information Technologies* 23.6 (2018), pp. 2737–2764.

[16] J.-L. Sierra, A. M. Fernández-Pampillon, and A. Fernández-Valmayor. “An Environment for Supporting Active Learning in Courses on Language Processing.” In: *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’08. Madrid, Spain: Association for Computing Machinery, 2008, pp. 128–132. ISBN: 9781605580784. DOI: `10.1145/1384271.1384307`.

[17] The Etherpad Foundation. *Etherpad Website*. `https://etherpad.org/`. Accessed January 14, 2021. 2021.

[18] R. Wilhelm, H. Seidl, and S. Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.