Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Visualization tool for teaching The Principles of Compilers

Bachelor Thesis

2014 Peter Dolák

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Visualization tool for teaching The Principles of Compilers

Bachelor Thesis

Study programme:	Computer Science
Study field:	2508 Computer Science, informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Richard Ostertág, PhD.

Bratislava, 2014
Peter Dolák

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Peter Dolák
Study programme: Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study: 9.2.1. Computer Science, Informatics
Type of Thesis: Bachelor´s thesis
Language of Thesis: English
Secondary language: Slovak

Title: Visualization tool for teaching The Principles of Compilers

Aim: The goal of this work is to implement visualization of algorithms used for teaching compilers principles.
After entering context free grammar, following features will be available to the user:
* FIRST and FOLLOW set calculation
* goto and closure function calculation
* grammar transforms (removal of left recursion, left factorization, ...)
* construction of parsing tables for LL, SLR, CLR, LALR parsers
* visualization of these tables by graphs
* visualization of parsing for chosen input

The result will be in form of web application implemented using HTML, JavaScript, SVG.

Supervisor: RNDr. Richard Ostertág, PhD.
Department: FMFI.KI - Department of Computer Science
Head of department: doc. RNDr. Daniel Olejár, PhD.

Assigned: 01.10.2013

Approved: 09.10.2013 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

62049539

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Peter Dolák
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Visualization tool for teaching The Principles of Compilers
Vizualizačný nástroj pre výuku princípov kompilátorov

Cieľ: Cieľom práce je implementácia a vizualizácia algoritmov používaných pri tvorbe kompilátorov.
Výsledkom budú webové stránky, ktoré budú slúžiť ako pomôcka pri výučbe tvorby kompilátorov.

Po zadaní bezkontextovej gramatiky bude možné napríklad:
* vypočítať množiny FIRST a FOLLOW
* vypočítať funkcie goto a closure
* transformácie gramatík (odstránenie ľavej rekurzie, lavá faktorizácia, ...)
* skonštruovať parsovacie tabuľky pre LL, SLR, CLR, LALR parser
* vizualizovať tieto tabuľky pomocou grafu
* vizualizovať parsovanie zvoleného vstupu

Pre implementáciu bude použité HTML, JavaScript, SVG.

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 01.10.2013

Dátum schválenia: 09.10.2013 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent vedúci práce

Acknowledgement

I would like to thank my supervisor RNDr. Richard Ostertág, PhD. for his help and advices, and also Mgr. Marek Zeman for his guidance in a former project that served as a basis for creating this thesis. Finally, I thank my family for their support.

Abstract

In this thesis we attempt to faciliate the study of syntax analysis, an important topic in the theory of compilers, which can be difficult to properly understand without good examples. As the main result of this thesis, we have created a web application allowing visualization of several algorithms for context-free grammar transformation, parse table creation and parser simulation. The thesis describes the development of a visualization framework and the implementation of these algorithms.

Keywords: compilers, syntax analysis, parsing, algorithm visualization

Abstrakt

V tejto práci sa pokúšame uľahčiť štúdium syntaktickej analýzy, dôležitej témy v teórii kompilátorov, ktorá môže byť náročná na správne porozumenie bez dobrých príkladov. Hlavným výsledkom práce je webová aplikácia umožňujúca vizualizáciu niekoľkých algoritmov pre tranformáciu bezkontextových gramatík, tvorbu parsovacích tabuliek a simuláciu parsera. Práca popisuje vývoj frameworku pre vizualizáciu a implementáciu týchto algoritmov.

KĽÚČOVÉ SLOVÁ: kompilátory, syntaktická analýza, parsovanie, vizualizácia algoritmov

Contents

Introduction

One of the most significant practical uses of the theory of formal languages is found in the process of compiling programming language code. Such as human languages cannot be purposefully translated word by word, also programming languages use their syntax to express critical information about the meaning of a program. In order for a compiler to translate between languages that usually have major differences in their syntax (most of the time we translate from a complex language to a simpler), it typically needs to convert the input string of characters (or a little more abstract entities – tokens) into a structure that is far easier to work with, a syntax tree. This can be achieved by determining how can the input string be generated by a context-free formal grammar describing the input language. This process is referred to as the syntax analysis or parsing.

Vast research was done in this area and methods were invented that allow parsing for large classes of context-free grammars in the optimal linear time. Depending on the power of such a class, these methods become increasingly complicated and may require somewhat larger effort to properly understand.

This work aims to provide help with teaching The Principles of Compilers and with self-study of this important topic. The main task is to create a client-side web application (powered by JavaScript) that implements several algorithms used to create a parser, allowing the user to observe their work on a chosen grammar using a step-by-step visualisation. Extra formatting is used to help the user see the changes performed in each step and text description is supplied to clarify their meaning. The tool can also simulate the parser itself and visualize some additional algorithms that can be performed on grammars.

The thesis begins with a talk about the motivation behind this project, its goals, used technologies and a comparison with some other projects serving a similar purpose. The second chapter describes the library for representing various formal language theory structures created for the needs of the project. A framework for step-by-step visualization of algorithms was created as well and will be discussed in the third chapter. Then follows a list of all the algorithms implemented in this project, providing definitions where necessary. The thesis also includes a usage example with screenshots from the application and concludes with a short talk about the future plans for extending it.

Chapter 1

Project background

1.1 Motivation

Most of today's parsers are created using automated tools that let the user specify an input grammar and produce an effective parser without exposing any of this task's true complexity. Nevertheless, Syntax Analysis is an important topic in the theory of compilers, formal languages and other areas of computer science. Proper knowledge of the employed methods is not just a matter of some general insight or only useful for authors of the mentioned automated tools. In many cases this knowledge will help us to choose a well-suited grammar, choose the right tool (depending on the generated parser type) and know what we can expect the tool to be able to do. Sometimes it may even enable us to create a better parser ourselves.

The algorithms used in the process of creating a top-down or especially a bottom-up parser (and the algorithm of the parser itself) are not trivial and can be much easier to understand and memorize when seen at work in a practical example. Hence the tool is created in such a way that it can be especially useful as an aid during a lecture. After a particular algorithm is explained, the tool can be used to demonstrate it on an example. Students can later work with the tool themselves for better understanding and memorizing of the algorithms, or to explore cases not covered on the lecture. A recommended approach would be trying each step on a piece of paper first and then checking with the tool. Finally, the tool can also be used to check students' answers on an exam. Of course, this tool could even be used for construction of parsing tables used in real world applications, although due to the algorithms being visualized, it can never reach the effectiveness of a standard parser generator.

1.2 Goals

Algorithms are visualized mainly by allowing the user to browse the computation on an example step by step and see the state of the grammar or various auxiliary structures..

This visualization attempts to achieve the following goals:

- The user is provided with various ways to enter the input context-free grammar that they would like to visualize the algorithms on
- The computation is organized into a hierarchical structure of steps that the user can easily navigate
- Since many algorithms make use of the result of another algorithm, the user has the opportunity to show such computations or skip them entirely
- Each step has at least a brief description of the action being performed, to aid with orientation in the visualized algorithm
- The changes made in a step are made clear by using appropriate highlighting, bringing the user's attention to semantically important changes

Some algorithms even visualize abstract structures, which are not physically represented during the execution of an algorithm, but can help understanding the semantics of the structures being computed upon.

1.3 Technology

JavaScript was chosen as the single programming language of this project. This choice allows for a high degree of portability and ease of access. Because no server-side computation is being used, the application can even be used without network access if preloaded or stored. Another advantage is the possibility of using modern web technologies for design – HTML, CSS3, SVG; speeding up the development.

The project heavily uses object-oriented programming, which JavaScript implements using the concept known as prototype-based programming. [MDNcOOP] This style of programming is quite flexible and most paradigms of classical object-oriented programming can be reproduced.

JavaScript libraries used by the project are following:

- **jQuery**
  A popular library greatly simplifying DOM Tree manipulation and extending the language with several useful utilities

- **Viz.js**
  Essentially the Graphviz library compiled to JavaScript, used to create the layout for directed graphs and render the graph to SVG

- **dagre-d3**
  A lightweight library for creating layouts of directed graphs, using the **D3.js** library to render the graph to SVG

- **lz-string**
  A lightweight compression library

1.4 Similar projects

The Context Free Grammar Checker [UCCFGC] is a web application created at the Department of Computer Science, University of Calgary. This tool enables the user to check certain properties of a user-specified grammar, generate various parser tables and apply many context-free grammar transformations. It is quite rich in functionality and helpful as a teaching aid in the area of syntax analysis because of displaying the output structures and automata, however none of the algorithms is visualized. It also cannot simulate a parser and the computation is done server-side.

Grammophone [DaiGram] is a web application created by Michael Daines. It is intended as a new version of The Context Free Grammar Checker and it implements a similar set of functionality. The most important change is that there are no active server side components. Grammar transformations are semi-automatic – the user is often given the choice on which grammar nonterminals or productions should a change be applied.

JFLAP [RodJFLAP] is a Java application created by Susan H. Rodger of Computer Science Department at Duke University. This very feature-rich application is able to visualise various algorithms from the area of formal languages, not omitting syntax analysis. For instance, it allows for visualising the construction of LL(1) and SLR(1) parsers and can simulate them as well, CYK parsing and Chomsky's normal form are also supported. Support for LALR(1) and CLR(1) parsers and some other grammar transformations is missing.

	This project	CFG Checker	Grammophone	JFLAP
Technology	web	web	web	Java
Client/server-side	client	server	client	client
Grammar transformations				
Reachable & generating	■	■	■	□
Epsilon-free	■	□	■	□
Chain-rule-free	■	□	■	□
Chomsky normal form	■	□	□	■
No left recursion	■	■	■	□
Greibach normal form	■	□	□	□
Left factoring	■	■	■	□
Transformations visualized	■	□	□	■
LR/LL parse table construction				
FIRST & FOLLOW	■	■	■	■
SLR	■	■	■	■
CLR	■	■	■	□
LALR	■	■	■	□
LL	■	■	■	■
Automaton graph displayed	■	■	■	■
Construction visualized	■	□	□	■
LR/LL parser simulation				
Parser simulated	■	□	□	■
Simulation visualized	■	□	□	■
CYK				
CYK parsing	■	□	□	■
CYK parsing visualized	future	□	□	■

Table 1.1: Project comparison. Note that the table only shows functionality implemented or planned in this project, all the other tools provide some functionality this project does not.

Chapter 2

Formal structure library

Before any algorithms could be visualized, it was necessary to implement good JavaScript representations for several structures from the theory of formal languages. A library was created providing rich classes for such structures, most importantly grammars and other outputs of the chosen parsing algorithms. The interfaces were designed with the goal of enabling us to efficiently express these algorithms using code of good readability (to the extend limited by the JavaScript programming language), heavily employing object-oriented programming, callback iterators and other patterns. This chapter mentions most of the classes and very few of their methods, the knowledge of which is essential for working with the library.

Certain characteristics are shared across many of the implemented structures. All of the object representations implement a `toElement` method creating its jQuery element set visual representation and `toString` method for a unique string representation. A `toData` method is available to obtain a representation of the structure that can be serialized as JSON or used to create a copy of the object. Several structures also feature a `highlight` method allowing to add certain highlighting visible in the element representation.

2.1 Grammar-related structures

Without a doubt the most important structure to be represented by the library is a context-free formal grammar. Following its usual definition as in [ALSU06, section 2.2.1], a context-free grammar consists of a set of terminal symbols (further referred to as terminals), a set of nonterminal symbols (or nonterminals), a set of production rules (or rules, productions) and a designation of one of the nonterminals as the start symbol. The object representation of a grammar is created by composing object representations of simpler entities.

2.1.1 Symbols

The most basic entities represented by an object are both types of symbols. Classes `Terminal` and `Nonterminal` inherit from a common class `Symbol`. The permissible text representation of a symbol is a string consisting of certain allowed characters (see table 2.1). Optional indexing of symbols is possible – any substring following the first occurrence of a slash character is visually formatted as a subscript. Terminal and nonterminal symbols are distinguished using the first character of their text representation – nonterminals start with an uppercase letter, any other first character designates a terminal. Name `eps` is reserved.

	letters		digits	special symbols
	uppercase	lowercase		
	A-Z	a-z	0-9	_ + - * / ( ) [ ] ' : ; ! ? @ $ #
when first:	nonterminal	terminal		

Table 2.1: Symbol text representation allowed characters

2.1.2 Words, rules & LR items

A sequence of symbols of either type creates a word, represented by the class `Word`. The text representation is composed by joining text representations of symbols with spaces. A special case is the empty word, which is represented in the user interface as `eps`. The internal representation of any word is an array of references to symbols; those are shared across all words defined within the context of a grammar. Some of the most useful methods include a callback iterator (method `forEachSymbol`) and basic regular expression matching (method `match`) - regular expressions can be used to match words over the alphabet $\{N, T, S, R\}$, where the letters substitute any nonterminal, any terminal, the grammar's start nonterminal and any nonterminal except the start, respectively.

The class `Rule` represents a context-free grammar production rule consisting of a left side nonterminal and right side word. The rule object simply holds references to such objects, which can be retrieved by methods `getLeft` and `getRight`. The following is an example of using the above mentioned `match` method to find out if a given production rule is regular:

Listing 2.1: `match` method usage example

```
if(rule.getRight().match("T*N?")) {
  // rule is regular
}
```

Another relevant class, `LRItem`, represents an LR(0) or LR(1) item (a rule with a dot at some position of the right side, [ALSU06, section 4.6.2]). An instance of this class holds a reference to a grammar's production rule, a numerical index determining the dot's position and an array of LR(1) lookahead symbols. In case an LR(0) item is desired, this array is simply left empty or omitted in the constructor call.

2.1.3 Grammars

A context-free formal grammar is represented by the `Grammar` class. The terminal and nonterminals sets are implemented as mappings of symbol text representations to symbol references. Method `tryGetSymbol` can be used to retrieve a symbol by it's text, methods `forEachTerminal`, `forEachNonterminal` and `forEachSymbol` allow callback iteration. The set of production rules is implemented as a mapping of left side nonterminal texts to rule groups, these are mappings of rule right side texts to actual rule object references. This approach allows for effective implementation of methods `forEachRule` and `forEachRuleWithLeft`, the latter of which will only iterate over the grammar's rules with a given left side nonterminal.

2.1.4 Grammar input/output methods

As one of the goals is to let the user choose the grammar used in the implemented algorithms, it is important that they can enter the grammar with ease. The project provides one graphical input method and four text input/output methods for grammars to meet the needs of various users and situations.

- **Rich input**
  Interactive graphical grammar editor, resembling a formal description of grammars. Symbols and rules can be created using buttons placed where the user would expect the new entity to be displayed; creating either a new nonterminal, terminal, adding a new rule to an existing group with a common left side, or creating a new rule group. Clicking on existing entities allows to change them (this also allows for symbols to be quickly renamed), clearing the input bow deletes them. Symbols need to be declared before they are used in productions. This input method provides best readability and is best suited for new users not familiar with the text input methods and for small changes to the currently used grammar.

- **Math-like text input**
  Text-only version of the above method. Symbols again need to be explicitly declared. Best suited for users familiar with the rich input method, especially when making

significant changes to the currently used grammar or when the application is being used on a low-end device.

- **Brief text input**
  Format used by The Context Free Grammar Checker and the Grammophone, therefore is best suited for users familiar with these applications. Since symbols are declared implicitly and only rules are specified, this is the fastest input format for specifying new grammars.

- **JSON**
  JSON parsable format. Useful when scripting is used to generate or further process the grammar.

- **HTTP link**
  The grammar is serialized, compressed, base64-encoded and included in the hash fragment of an HTTP link to the application. The grammar is automatically used if the application is opened with such a link. This method is intended for sharing the grammar using the Internet (for example the link could be included in a course homework assignment).

2.2 Algorithm-related structures

Most of the algorithms implemented in this project make use of other structures than just the grammar. Algorithms like form conversions can sometimes use important auxiliary structures that should be visualized to the user as well (mainly symbol sets and tables). Moreover, all of the parsing algorithms yield such other structures as their result (mainly tables, automata and parse trees).

Later we will explain that the algorithm framework needs to be able to make snapshots of all the visualized structures in each step. Since in any of the algorithms there is always exactly one grammar present, it is acceptable to pay some special attention to it and create its copies by directly referencing the `Grammar` class. The other structures, however, are used differently by each algorithm, therefore a more general approach is necessary.

2.2.1 Common structure mechanisms

All of the structures provided by the library except the grammar inherit from the `Structure` base class. They need to implement the following methods:

- `toStructureInnerElement` – should return a jQuery element set with all the structure dependent markup

- `getType` – returns a string with the name of the subclass (JavaScript doesn't provide any reliable built-in method of doing that)
- `toConcreteData` – should return serializable object with all of the structure's important state variables
- `fillStructureData` – should replace the current structure's state using a data object generated by the above method
- a constructor delegating the `id` and `name` arguments to the `initStructure` method inherited from the base class

Correct implementation of those methods will then allow the proper function of the following methods inherited from the `Structure` base class:

- `toElement` – returns a jQuery element set with the full structure markup, displaying a name if it was set
- `toData` – extends the concrete data object with other information needed to reconstruct the structure
- `name`, `id` – getter methods for the name of the structure displayed to the user and its internal identifier
- `priority` – getter/setter method for a structure's priority; the lower this number is, the higher is the structure's element positioned in the visualization; the grammar's implicit priority is 0
- `Structure.fromData` static method – creates a new structure of any type using the provided data object
  The specialized structure's constructor is dynamically located in the document object model using the class name string provided by `getType`, the newly created structure is then initialized using the `fillStructureData`.

We have to state that although this approach to creating copies of structures works, some downsides became apparent later. The first one is the mere fact that a structure needs to manipulate all its state data in the `toConcreteData` and `fillStructureData` methods. Aside from being a burden, this can lead to very obscure bugs if the programmer starts using a new state data field and forgets to modify those methods. Another downside is performance – creating the state's data representation has the complexity of a deep copy. In the future, this could be partially addressed by using immutable objects only.

2.2.2 Set-like structures

The abstract class `SetStructure` serves as a basis for implementing representations of set-like formal structures. A derived class will allow to create sets over a certain domain of elements, which are typically simpler objects not derived from the `Structure` base class. Such objects should override their `toString` method to return their unique text representation. The derived class should then implement the `createSetItemFromText` method to use a given text representation of an element to return an appropriate object for it. A grammar object is also passed, so that instead of creating new instances, references can be returned where possible. Concrete structures derived from the `SetStructure` class include:

- `SymbolSet` – elements of this set are the symbols of a grammar, or an empty word
- `LRItemSet` – elements are LR items for a grammar
  The structure `LRState` representing a state of an LR automaton is essentially a wrapper for two `LRItemSet` structures storing core and non-core LR items, all represented by the `LRItem` class.
- `ParserActions` – elements are actions or parse table entries of an LR/LL parser, these are:
  - `LRActionShift`, `LRActionReduce`, `LRActionAccept` – LR actions Shift, Reduce & Accept; for the LR ACTION table
  - `LRGotoEntry` – an entry in the GOTO table
  - `LLActionProduce` – an entry in the LL parse table
  - `ParserActionError` – a user-defined error in the LR/LL parse table

2.2.3 Table structures

The class `TableStructure` provides representation for formal structures that can be best expressed as tables. Rows and columns (with names used in header cells) can be declared during the construction or added and removed dynamically. Cells of such a table are usually set structures; specialization for an instance is achieved by passing the structure's class object in `TableStructure`'s constructor.

One important use of the `TableStructure` class is representing an LR/LL parse table. In this case, each cell of the table is occupied by an instance of the `ParserActions` set structure. This allows each parse table cell to have zero to many parser actions defined. When a computed parse table has conflicts (multiple actions), a special parse table input method can be used to disable an action and resolve them. This is done by clicking on the action, toggling the enabled/disabled state.

2.2.4 Parser structures

The following structures are used by LL and LR parser algorithms.

The input word for a parser provided is represented by the class `ParserInput`. Before a parser simulation algorithm is computed and visualized, the user is prompted to specify an input word; that is stored, accessed and visualized by this structure.

Both parser algorithm make use of a stack, represented by the class `ParserStack`. Elements inserted into the stack should be structures derived from the `Structure` base class, requiring symbols to be wrapped using the `SymbolWrapperStructure` class. The contents can be manipulated using methods similar to those typical for a stack (push, peek, pop).

Finally, both parser algorithms emit a sequence of production rules as their output, which is collected by the `ParserOutput` class. This structure is typically not visualized until the parser algorithm execution finishes successfully. At this point, when the structure should be visualized, the rules are used to construct a parse tree, which is visualized.

2.2.5 LR automaton

The most complex formal structure found in the project is the LR automaton, represented by the `LRAutomaton` class. The automaton is composed of an enumerated list of states (item sets, using the above mentioned `LRState` class) and a set of transitions between them (implemented as ordered pairs over the set of state numbers). The automaton has to be able to efficiently look up whether a given item set is already an existing state, and add it if necessary; transitions are added dynamically as well.

With the states being nodes and transitions being edges, the automaton is depicted as a directed graph, potentially with cycles. Visualizing such a graph is not a trivial problem and external libraries were used to calculate the graph's layout. The user can choose one of these three options to change the functionality of an automaton's `toStructureInnerElement` method:

- **None**
  With this option the LR automaton structure is not visualized at all. Very large graphs are not readable using any of the below mentioned methods and the execution time can significantly complicate browsing the algorithm visualization; in such case it is advisable to use this option.

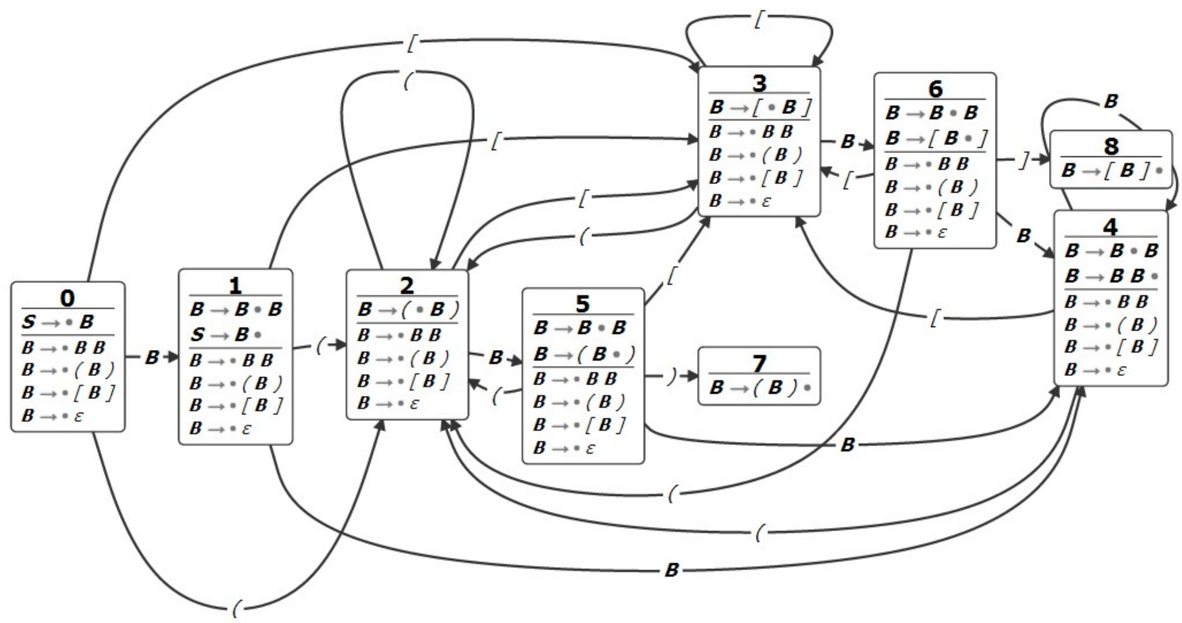

Figure 2.1: Example of using dagre-d3 to visualize the SLR(1) automaton graph for the grammar { $B \to BB \mid (\, B \,) \mid [\, B \,] \mid \varepsilon.$ }

- **dagre-d3**

  Dagre is a JavaScript library that allows to easily lay out directed graphs on the client-side. The dagre-d3 library acts a front-end to dagre, using the D3 library to provide actual rendering of the layout to SVG. HTML labels for nodes and edges are natively supported.

  Creating the layout and markup for the automaton with dagre-d3 is done in the `toDagreGraph` method. Using this library was very straightforward and no intervention with the generated markup was required. The minified sources of the dagre-d3 and D3 libraries are less than 200 kB in size.

  In smaller graphs the layout of nodes created by dagre is logical and well organized, but the overall quality deteriorates with increasing amount of edges. Edges may often overlap in angles that make them difficult to follow, all loops are laid out poorly. The execution is moderately resource consuming.

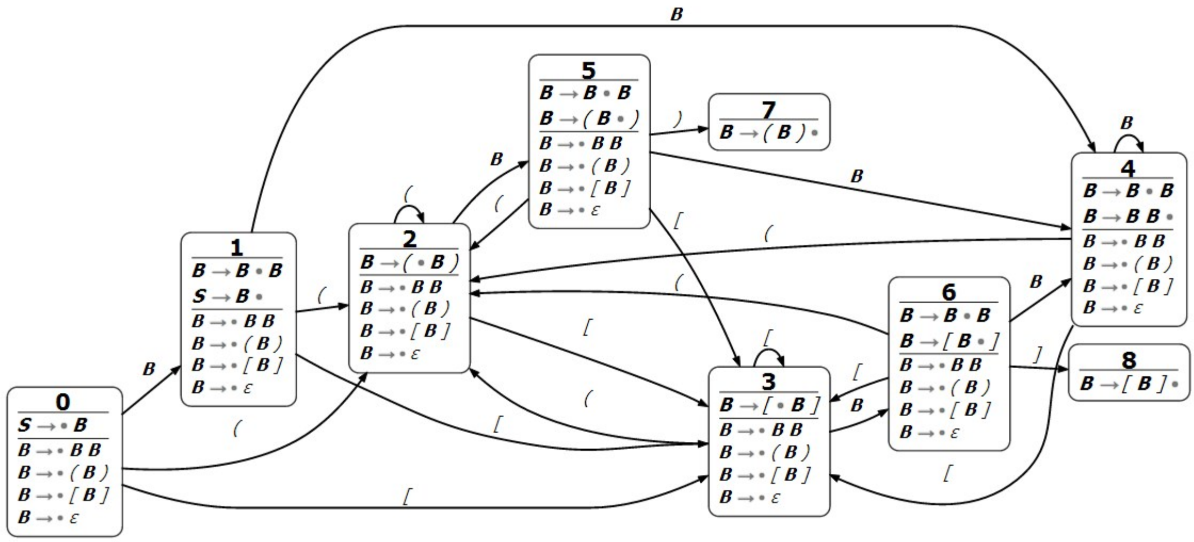

Figure 2.2: Example of using Viz.js to visualize the SLR(1) automaton graph for the grammar { $B \to BB \mid (B) \mid [B] \mid \varepsilon.$ }

- **Viz.js**

  This library was created by compiling the GraphViz library to JavaScript using the Emscripten source-to-source compiler, allowing for layouts to be created client-side. It uses the powerful DOT language to input graphs and allows to choose one of many output formats, including SVG. There is no native support for HTML labels.

  Method `toGraphvizGraph` creates the layout and markup for the automaton with the help of Viz.js. To achieve HTML labels, the dimensions of label elements for nodes and edges need to be pre-calculated and passed in the DOT language input string. Afterwards, the SVG is traversed and HTML label elements (wrapped in the `foreignObject` SVG element) are used to replace some existing SVG markup. The size of the minified source for Viz.js is above 1.3 MB.

  Viz.js is able to keep the layout readable even for somewhat larger graphs than dagre. The main drawback is the resource consumption, which grows quickly with the size of the graph; the execution is generally slower than dagre.

Chapter 3

Algorithm visualization framework

Using the formal structure library, algorithms can now be easily implemented. To visualize their step-by-step implementation, however, requires significantly more effort. A powerful framework for visualization of algorithms was created in this project; this section should describe its goals, application programming interface with an example algorithm implementation, the concepts behind this framework and the user interface.

3.1 Goals

We will now analyze the goals set for the visualization, so that it is apparent what aspects common for any algorithm visualization should be separated into a framework.

- **Freely navigable visualization**
  To achieve a step-by-step visualization, certain points should be set in the algorithm's execution, the immediate state of which can be later viewed by the user. We will be referring to these points as steps. As we would like to impose no restrictions on the order in which these steps can be displayed by the user, the execution will be precomputed and a snapshot of the state in each step will be stored. Such a snapshot should allow the complete reconstruction of the original object structure so that it can be easily visualized.

- **Hierarchical organisation of steps**
  If the steps were to be organized in a strictly linear fashion, the visualization would become less useful for complex algorithms with many steps. We will enforce a hierarchical organisation of steps based on their semantics for reasons such as:

  - **providing the user with more context** – a look upwards in the hierarchy can quickly answer the question why is a certain subtask necessary

  - **more flexible navigation** – the user can easily skip repetitive groups of steps that will not aid in understanding the algorithm; or more easily locate groups of steps that can
  - **better memorability** – information organized in a clear and logical hierarchy is easier to remember than when in a sequence [MM97]

  To our benefit, the structure of steps will come naturally from the structure of the code. Following good coding practices, the implementation of an algorithm shall be separated into procedures that perform well-defined tasks and may rely on other procedures to carry out any subtasks. The framework can, in a sense, trace the calls to those procedures to gain the information needed to create the step hierarchy.

- **Delayed step execution**
  As a downside of JavaScript's event loop and its run-to-completion model [MDNcEL], long execution times of event loop messages (synchronously executed code blocks) cause browser freeze-ups and other unpleasant effects. Where necessary, this is treated by cutting the down large messages to several smaller ones. As the execution time of a visualized algorithm can easily become noticeable with large inputs, the algorithm framework should only carry out a finite amount of procedures in one message. This should be achieved by executing a procedure's subtasks asynchronously - instead of actually calling a subprocedure, it is merely scheduled to be carried out after the procedure's own code.

- **Flexibility**
  The hierarchical structure of steps will not only be shaped differently for different algorithms, but even for executions of the same algorithm on different inputs. For instance a parser algorithm's step may use diverse subtasks depending on the action defined in its parse table. The framework should enable the algorithm to quickly and easily adjust its program flow depending on the information it gains. It should allow some typical programming constructs, like conditionals and loops, to be emulated at the level of steps.

3.2 Framework interface

Implementing a new visualized algorithm is made fairly simple by the framework. A new JavaScript function with an empty body should be declared (we will refer to this function as the algorithm object). Then the function `setupAlgorithmImplementation` is called, passing the algorithm object and another object used as a dictionary to set some configuration values (see Listing 3.1 for an example). The mandatory values are:

- `id` – string with the identifier of the algorithm; by convention should be the same as the function's name
- `name` – string with a short name of the algorithm used in the menu
- `type` – determines where the algorithm should be displayed in the menu; either `"FormConversion"` (also declares that the algorithm implements grammar form checking, see below) or `"ParsingAlgorithm"`

After the algorithm object is set up, task procedures should be defined as methods of this function object (set to the object as its direct properties). Each time such a method is executed, a new step is generated in the visualized algorithm execution hierarchy. The method accepts one parameter – a controller object, which is also accessible via the `this` keyword. The whole body of this method will be executed in a single step. All interaction with the framework is done using the controller object, which provides the following methods:

- `myNameIs` – This method accepts a string which will be used as the description of the current step displayed to the user. Since the description can be composed and set anytime during the step, it is possible to provide very specific information about what is being done and why. It is strongly recommended that the description string is composed using the `_` (underscore) method, which provides very basic templating functionality. Occurrences of `%s` in the format string passed as the first parameter are replaced by strings passed as other arguments, in their respective order.
- `grammar` (alias `g`) – Returns a reference to the step's grammar, which is actually an instance of the `ConversionGrammar` class. This is a `Grammar` subclass, extending it with methods that allow highlighted modifications of the underlying formal structure.
- `structures` (alias `s`) – Returns an instance of the `Structures` class. This serves as a collection of any structures that the algorithm uses, allowing new structures to be declared, existing structures to be retrieved or removed.
- `data` (alias `d`) – Returns the reference to an object that is shared across the whole algorithm's execution. It can be used for storing any auxiliary variables used by multiple steps. The contents of this objects are not visualized, thus it should only be used for storing parts of the execution's state that are obvious to the user.
- `task` – This method accepts a reference to another task method, which is scheduled for execution as a subtask. The subtask will have its own step in the visualization and will be located under the step of the invoking task in the hierarchy. The code of a scheduled subtask is not executed immediately – all code of the invoking task is executed before

anything else. Then the code of the first scheduled subtask is executed, also executing any subtasks of its own. Then the second directly scheduled subtask is executed, and so on.

- `willRepeat` Calling this method in the body of a task method will make sure the same task is executed again right after all the scheduled subtasks. The current task will thereby become another subtask of its parent task, executed before its further subtasks.

- `hook` – Similarly to `task`, this method accepts a reference to a JavaScript function. However, this method allows scheduling only a single function, which will be executed immediately after the rest of the task method's body; and it will not create another step. Setting such a "hook" can be especially useful because the changes done by its code will not be visualized in the current step, but will have already taken effect in the next one.

The execution of an algorithm begins at the task method `main` of the algorithm object. Since it constitutes the root of the hierarchy, the `willRepeat` controller method cannot be used there. The `task` controller method is not limited to task methods of the same algorithm object, this way other algorithms or their parts can be reused (which is done extensively in the project). The visualization does not actually allow navigating to the step represented by `main`, but an artificial step is created as its subtask to allow viewing the initial state. Furthermore, by convention there should be a step added as the last subtask of `main` that does not change the state and its description says that the algorithm has finished.

If the algorithm object was defined using `"FormConversion"` as the type, it should also have the method `check` defined. This method receives a grammar as its only argument and should return a boolean value indicating whether the grammar is already in the normal form the algorithm converts to.

Although this controller interface is quite simple, it proved to be comfortable to use and sufficient to implement all of the algorithms visualized in this project. Even so, as the mechanisms behind the interface are very powerful, other interface methods providing even more flexibility can be easily added in the future.

3.2.1 Algorithm implementation example

In the following example we will implement a trivial algorithm for converting a grammar to a normal form without loops. A loop is a production rule of the form $A \to A$, where A is a nonterminal. Clearly, such productions have no effect on the language generated by a grammar and can be removed.

The implemented algorithm will remove one loop per step and will also visualize an incrementally constructed set of nonterminals, for which a loop existed in the original grammar. In the actual implementation, the list of all loops is collected in the method `main`, so that the grammar productions do not have to be iterated over in each step.

Listing 3.1: example of implementing conversion algorithm for removing loops

```
function NoLoopsForm() { }

setupAlgorithmImplementation(
    NoLoopsForm,
    {
        id: "NoLoopsForm",
        name: _("no loops"),
        type: "FormConversion"
    }
);


NoLoopsForm.main = function (c) {
    // set the description
    c.myNameIs(_("remove loops"));

    // declare a new SymbolSet structure
    // stores symbols that had loops, but they were already removed
    c.structures().newSymbolSet('hadLoops', _("Had loops"));

    // create a queue with the loops to be removed (not visualized)
    var queue = c.data().loopQueue = [];

    // inspect all the productions to find loops (callback iterator)
    c.grammar().forEachRule(function (rule, leftNonterm, rightWord) {
        if (
            rightWord.match("N") // the rule is in the form A -> B
            && leftNonterm.equals(rightWord.get(0)) // A = B
           )
            // add the rule to queue
            queue.push(rule);
    });

    // if the queue isn't empty, we remove the loops in substeps
    // we only schedule the first substep, which will be repeated
    if (queue.length > 0)
        c.task(NoLoopsForm.removeSingle);

```

```
    // we also schedule a clean up task
    c.task(NoLoopsForm.done);
}

NoLoopsForm.removeSingle = function (c) {
    // retrieve the symbol set structure and the loop queue
    var hadLoops = c.structures().getSymbolSet('hadLoops'),
        queue = c.data().loopQueue;

    // destructively retrieve a loop production
    var loop = queue.shift();

    // generate the description
    c.myNameIs(_("remove the production %s -> %s",
        loop.getLeft(), loop.getLeft()));

    // add the nonterminal to the set
    hadLoops.add(loop.getLeft());

    // remove the production from the grammar
    c.grammar().taskRemoveRule(loop);

    // repeat this step until the queue is empty
    if (queue.length > 0)
        c.willRepeat();
}

NoLoopsForm.done = function (c) {
    c.myNameIs(_("done"));
}


NoLoopsForm.check = function (grammar) {
    var ret = true;
    // search the grammar for loops, if found, check is negative
    grammar.forEachRule(function (rule, leftNonterm, rightWord) {
        if (
            rightWord.match("N") // the rule is in the form A -> B
            && leftNonterm.equals(rightWord.get(0)) // A = B
           )
            return ret = false;
    })
    return ret;
}
```

3.3 Framework mechanics

3.3.1 Algorithm controller tree

In order to achieve the goals set for the framework, we have designed a data structure that controls the execution of an algorithm, stores the state of each step and later allows to browse these states to visualize them. The structure is in its essence a rooted tree and we will be referring to it as the **algorithm controller tree**. Each node of this tree is equivalent to a single step of the algorithm's execution; any subtree is equivalent to a step and all its substeps. The following is the functionality provided by any algorithm controller tree node, represented by the class `ACTNode`. It should be noted that this structure is not actually exposed by the framework.

- **Navigation**
  A node holds up to five references to surrounding nodes in the tree to allow quick navigation according to the needs of the framework:
  - to the node's parent
  - to the node's first and last child
  - to the node's previous and next sibling

  These five references were sufficient for an efficient implementation of bidirectional prefix order traversal of the algorithm controller tree.

- **Procedure code**
  The only argument passed to a newly created `ACTNode` is a reference to some JavaScript function, which the framework will later execute in order for the step to be carried out.

- **State**
  A node also holds the state of a step, consisting of references to `ConversionGrammar` and `Structures` class instances (see above). Note that the auxiliary data object exposed by the framework is not stored per node, as it is not visualized. A node's state must be initialized before the attached code is executed; afterwards the new state (modified in the execution) can be retrieved.

- **Name & numbering**
  When browsing the execution tree of an algorithm, a node's name will be displayed to the user as the step's description. Each node is also assigned a numbering string (composed as the numbering of the parent node and the child's index).

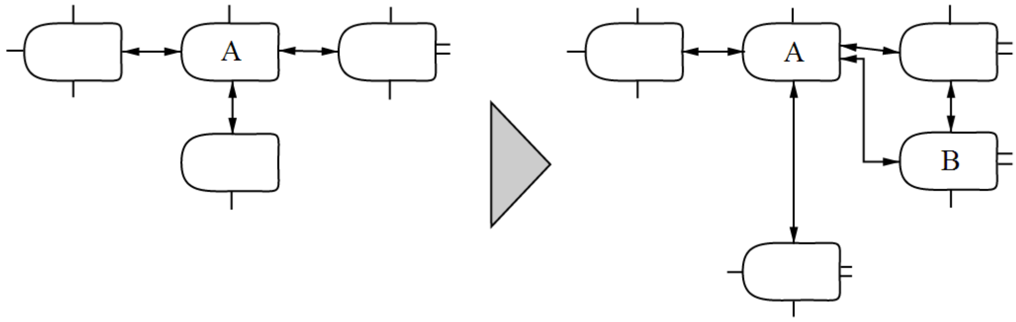

Figure 3.1: Class `ACTNode` node adoption example; `A.adopt(B)`

- **Adding nodes**

  New nodes can be added to the tree by calling one of the following two methods on any existing node.

  - **adopt** – `A.adopt(B)` will add the node `B` as the last child of the node `A`
  - **duplicate** – `A.duplicate()` will create a duplicate `B` of the node `A` and add it as the last child of `A`'s parent node (the root node cannot be duplicated)

  These methods will modify all the references to surrounding nodes used for navigation accordingly.

3.3.2 Algorithm controller

The Algorithm controller tree is used by the `AlgorithmController` abstract base class, arguably the core of the algorithm framework. Implementations of actual algorithms are created by extending this class (which is done by the `setupAlgorithmImplementation` helper function), the controller object passed to task method is an instance of the derived algorithm class. There are three phases of this instance's life cycle: initialization, execution and browsing.

When the instance (controller object) is initialized, the root node for a new Algorithm Controller Tree is created, using the algorithm's task method `main` as the initial procedure. It also receives its initial state – usually the currently active grammar and an empty collection of structures. Furthermore, an empty object is created for any auxiliary data used by the algorithm and stored in the controller.

Soon after the initialization, the controller receives a signal to start executing the algorithm. The root node is set as the active node and the initial call to the recursive procedure `step` is made. This procedure firstly executes the code attached to the active `ACTNode` object (potentially adding new nodes to the tree) and retrieves the modified state of the algorithm's execution. It is then attempted to locate the node that follows the active tree node in prefix traversal order.

If such a node exists, it is set as the new active node. It also receives the finalized state of the previous node. The procedure `step` will then recursively call itself to carry out the next step. Depending on how many nodes were already processed, this recursive call is either done in the same JavaScript event loop message (using a direct function call), or in a new message (using JavaScript's `setTimeout`). This achieves that only a finite amount of step executions is carried out within one message and the event loop does not become frozen.

If no next node was found, the algorithm is considered finished. The controller then collaborates with an instance of the `AlgorithmController` class, which takes care of all user interaction while the algorithm is browsed. The controller object is destroyed once the user executes another algorithm, or makes changes to any of the structures used as the input.

3.4 Algorithm browsing user interface

After the execution is finished, the user can browse the algorithm's steps using two methods of navigation. The first method provides six buttons that navigate to one of the following:

- next step (▶)
- previous step (◀)
- next step, skipping substeps of the current step (▶▶)
- previous step, skipping substeps of the current step (◀◀)
- first step, initial state of the algorithm (▶|)
- last step, algorithm result (|◀)

The second method provides a selectable list of all steps, which is created by traversing the step hierarchy in prefix order.

While a step is displayed, the user can see all the visualized structures that the step's state consists of. The description and numbering of the current step and all the steps upwards in the hierarchy is visible as well.

If the user wishes to use the result of an algorithm for further computation in the application (a modified grammar or a generated parse table), they can either store the final state or store the state of any step of their choosing.

Chapter 4

Implemented algorithms

4.1 Grammar normal forms

Each grammar normal form defined in this section has an algorithm implemented in the project, which converts the input grammar to another equivalent grammar meeting its criteria. Aside from implementing and visualizing these algorithms, the project also automatically reports which of these forms does the currently displayed grammar already comply with.

As not all of these normal forms are known under the names used in the project, we provide their definitions. We also provide very brief explanations of how do the conversion algorithms operate.

4.1.1 Reduced form

We will say a context-free grammar is in the reduced normal form (or has no useless symbols, [HMU01, section 7.1.1]) if and only if all of the following is met:

- There is no production of the form $A \to A$, where A is a nonterminal (a loop).
- All nonterminals are generating. A nonterminal is generating if and only if there is any terminal word derived from the nonterminal.
- All nonterminals are reachable. A nonterminal is reachable if and only if it is contained in any sentence form derived from the start nonterminal.

After removing loops from the grammar, the conversion algorithm iteratively constructs the set of generating nonterminals and removes the rest, then the same is done for the set of reachable nonterminals.

4.1.2 Epsilon-free form

We will say a context-free grammar is in the epsilon-free normal form (or has no ε-productions, [HMU01, section 7.1.3]) if and only if all of the following is met:

- There is no production of the form $A \to \varepsilon$, where A is a nonterminal other than the start nonterminal.
- If the production $S \to \varepsilon$ is present, the start nonterminal S is not found in any production right side.

The algorithm first creates a set of nullable nonterminals (those that generate the empty word) and then adds subsequences of existing productions with those nonterminals skipped. ε-productions are removed, possibly except for $S \to \varepsilon$ (in such case a new start nonterminal might be necessary to enforce the second criterion). Thus, if a grammar used to generate the empty word, the converted grammar can still do so, however the presence of ε-productions is reduced to a single and well-maintainable case.

4.1.3 Chain-rule-free form

We will say a context-free grammar is in the chain-rule-free normal form (or has no unit productions, [HMU01, section 7.1.4]) if and only if there is no production of the form $A \to B$, where A, B are nonterminals. It also holds that if $A \to \alpha$ is a production that was added during the conversion, there must have been a rule with α as its right side in the original grammar.

For each nonterminal the algorithm finds the set of nonterminals reachable by chain rules, then it adds the chain-reachable productions to make chain rules useless.

4.1.4 Weak Chomsky form

We will say a context-free grammar is in the weak Chomsky normal form [RF13, section 3.3] if and only if each of its productions is in one of the following forms:

- $A \to B\ C$, where A, B, C are nonterminals
- $A \to t$, where A is a nonterminal and t a terminal
- $A \to \varepsilon$, where A is a nonterminal

Nonterminals for each terminal are created where necessary to make sure other productions use nonterminals only. Unit productions are prolonged using a nullable nonterminal. Long productions are iteratively decomposed until the grammar meets the form. This weak version cannot be used for CYK parsing.

4.1.5 Strict Chomsky form

We will say a context-free grammar is in the strict Chomsky normal form [HMU01, section 7.1.6] if and only if each of its productions is in one of the following forms:

- $A \to B\ C$, where A, B, C are nonterminals
- $A \to t$, where A is a nonterminal and t a terminal
- $S \to \varepsilon$, where S is the start nonterminal

Furthermore, if the production $S \to \varepsilon$ is present in the grammar, the start nonterminal S cannot be present in any production right side.

Nonterminals for each terminals are created where necessary to make sure other productions use nonterminals only. Then the epsilon-free and chain-rule-free normal form conversions are applied. Long productions are iteratively decomposed until the grammar meets the form. Grammars in this form again preserve generation of the empty word and can be used in CYK parsing.

4.1.6 Left recursion removal

We will say a context-free grammar has no left recursion [ALSU06, section 4.3.3] if and only if there is no nonterminal A such that $A \Rightarrow^+ A\ \alpha$, where α is any word over the grammar symbols. In words, a nonterminal cannot be used to derive a sentence form beginning with the same nonterminal.

The conversion algorithm applies the epsilon-free and chain-rule-free forms, afterwards the grammar nonterminals are enumerated. An iterative algorithm is used to enforce that if $N_k \to N_i\ \alpha$ is a production, then $i > k$. The algorithm then attempts to restore the original nonterminal names.

4.1.7 Greibach form

We will say a context-free grammar is in the Greibach normal form [RF13, section 3.6] if and only if each of its productions is in one of the following forms:

- $A \to tA_1A_2...A_k$, where t is a terminal and $A_1, A_2, ..., A_k$ for $k \geq 0$ are nonterminals
- $S \to \varepsilon$, where S is the start nonterminal

Furthermore, if the production $S \to \varepsilon$ is present in the grammar, the start nonterminal S cannot be present in any production right side.

The algorithm performs the above mentioned left recursion removal, in addition, all nonterminals at the beginning of a production are iteratively expanded.

4.1.8 Left factorization

We will say a context-free grammar is left factored [ALSU06, section 4.3.4] if and only if for each nonterminal A it holds that there are no two productions $A \to \psi\,\alpha$ and $B \to \psi\,\beta$, where ψ is a non-empty word and α, β are any words over the grammar's symbols. In words, two productions with the same left side nonterminal cannot have a non-empty common prefix in their right sides.

For each nonterminal, the algorithm iteratively searches for common production prefixes of increasing lengths. Productions with a common non-empty prefix are factorized using a new nonterminal.

4.2 Parsing

The following algorithms do not modify the grammar they receive as an input (or there is only an insignificant temporary modification) and produce a certain formal structure as their output. With the exception of CYK, all algorithms are visualized, including the structures they output.

As these algorithms are well known by their name used in the project, it is outside the scope of this thesis to define all the output structures they generate or provide in-depth explanations of their operation. This section states where in an algorithm's implementation do certain actions happen, however.

4.2.1 CYK

If a grammar is in the strict Chomsky normal form, the CYK algorithm (Cocke, Younger, Kassami) [HMU01, section 7.4.4] can be used to parse any input word. If a word was parsed successfully, the algorithm also allows to display a randomly chosen parse tree and its corresponding left-most and right-most derivations (note that multiple parse trees could have been found if the grammar is ambiguous). The algorithm's execution is not visualized, but it is planned to be in the future.

4.2.2 First & Follow sets

This algorithm computes the $FIRST$ and $FOLLOW$ sets [ALSU06, section 4.4.2] for each nonterminal of the input grammar. All these sets are visualized in a single table.

$FIRST(A)$ contains all terminals that can occur as the first symbol of a word generated by the nonterminal A. Furthermore, if A generates the empty word, we display ε in the set as well.

$FOLLOW(A)$ contains all terminals that can occur after the nonterminal A in a sentence form generated by the grammar. Furthermore, if the grammar generates a sentence form where A is the last symbol, we display an end marker ($) in the set as well.

The algorithm constructs these sets by repeatedly processing all the grammar's productions and adding terminals to the appropriate sets until no more can be added. The visualization skips processing of productions that have no effect.

4.2.3 SLR parse table creation

This algorithm generates the SLR(1) parse table for the input grammar, a single table structure composed of its $ACTION$ and $GOTO$ table. The LR(0) automaton is created in the process as well and is visualized as a graph. The SLR (simple-LR) method for constructing parse tables for an LR(1) parser is described in [ALSU06, section 4.6.4]; the surrounding sections of the book also explain the important concepts, like LR items and automatons.

The algorithm starts by computing the $FIRST$ and $FOLLOW$ sets for the input grammar. Then a new start nonterminal is created for the grammar (this modification is just temporary) to more easily describe the initial LR automaton state – if S is the original start nonterminal and S_0 is the new one, the initial state contains only the item $[S_0 \rightarrow \bullet\ S]$. States of the automaton are then processed iteratively. Firstly, the $CLOSURE$ operation is computed from all core items of a state. Then for each final item $[A \rightarrow \alpha\ \bullet]$ in the state an action is added to the parse table, ACCEPT if it is the item $[S_0 \rightarrow S\ \bullet]$, REDUCE otherwise. As there is no item lookahead, these actions are added for every symbol in $FOLLOW(A)$. Finally, transitions from this state are attempted for every symbol in the grammar and all new non-empty states are created. Each such a transition also adds a new record to the parse table, action SHIFT for a terminal, GOTO for a nonterminal. When all new states have already been processed, the algorithm stops, leaving a complete parse table and LR automaton. Conflicts (table cells with multiple actions defined) are highlighted.

4.2.4 CLR parse table creation

This algorithm generates the CLR(1) parse table for the input grammar, as described in [ALSU06, section 4.7.3]. The CLR (canonical-LR) method can avoid some conflicts by using LR(1) items and the states of an LR(1) automaton, allowing better use of the lookahead symbol while parsing.

The work of this algorithm is very similar to the generation of SLR tables. The initial LR(1) item is $[S_0 \rightarrow\ \bullet S,\ \$]$, where $ is the end marker lookahead symbol. The $CLOSURE$ operation takes the symbol $FIRST$ sets into account when creating new items. Actions REDUCE/ACCEPT rely on the lookahead of final items instead of the $FOLLOW$ sets.

4.2.5 LALR parse table creation

This algorithm generates the LALR(1) parse table for the input grammar, as described in [ALSU06, section 4.7.4]. The LALR (lookahead-LR) method makes use of LR(1) items just as CLR, but merges the states of the LR(1) automaton with the same core items, disregarding of their lookahead.

The project implements the space-inefficient algorithm that explicitly creates and then merges the states of an LR(1) automaton. The first part of the algorithm works exactly like the CLR parse table creation, then the algorithm merges the appropriate states iteratively – in each step a group of states is merged into a single one, adjusting the visualized parse table and LR automaton.

4.2.6 LR parser simulation

Once an LR parse table has been computed and stored, this algorithm can be used to simulate an LR parser [ALSU06, section 4.6.3] on any input word. The same algorithm is used for tables created by SLR, CLR and LALR methods. The simulation will be attempted even if the input parse table has conflicts.

Five structures are used in the process:

- **parse table** – consisting of the $ACTION$ and $GOTO$ table
- **parser input** – before the algorithm begins execution, the user is asked to enter an input word
- **parser state stack** – the main stack for storing parser states added by SHIFT actions and retrieved from the $GOTO$ table on REDUCE actions
- **parser symbol stack** – a supplementary stack for storing symbols (input terminals on SHIFT actions and production left side nonterminals on REDUCE actions); this structure is not actually necessary for the algorithm, but may aid understanding it
- **parser output** – if the parsing has finished successfully, it visualizes the generated parse tree

The simulation begins with the initial state in the main stack. Each move of the parser is represented by a step, which retrieves an action from the $ACTION$ table based on the top of the state stack and the input lookahead, and a few substeps dependent on the action retrieved. If multiple or no actions are found in a move, the algorithm fails.

4.2.7 LL parse table creation

This algorithm generates the LL(1) parsing table for the input grammar, as described in [ALSU06, section 4.4.3].

As in the case of LR parse table generation, the algorithm starts by computing the $FIRST$ and $FOLLOW$ sets for the input grammar. Then, each grammar production $A \rightarrow \alpha$ is processed in up to three steps. The first step calculates $FIRST(\alpha)$, the other two steps may add the production in certain columns of the A row of the parse table. In the second step, these columns are any terminals in $FIRST(\alpha)$. If $\varepsilon \in FIRST(\alpha)$, then in the third step, these columns are also any terminals or the end marker $ in $FOLLOW(\alpha)$.

4.2.8 LL parser simulation

Once an LL parse table has been computed and stored, this algorithm can be used to simulate an LL parser [ALSU06, section 4.4.4] on any input word. The simulation will be attempted even if the input parse table has conflicts.

The structures used by the algorithm are similar to the LR parser, but there is only a single **parser symbol stack**, for storing symbols of the used productions. The simulation begins with the start nonterminal in the stack. Each move of the parser is represented by a step trying to determine the next action based on the top symbol on the stack and the current lookahead. If multiple or no actions are found in a move, if the lookahead doesn't match a terminal on the top of the stack, or if the stack underflows without consuming all the input, the algorithm fails.

Chapter 5

Project usage example

This chapter contains an example of using this application with screenshots. The chosen example usage should well demonstrate the most important features of the application and provide a beginner user a good idea on how to work with it.

The task we will be trying to achieve is to parse the word "*if cond then if cond then st else st*" using LL parsing on a grammar demonstrating the dangling else problem [ALSU06, section 4.8.2].

We begin with opening the menu by clicking at the top bar and using the brief text input method (**grammar (text)** in the menu, then the tab **brief**) to enter a very simple grammar for the problem.

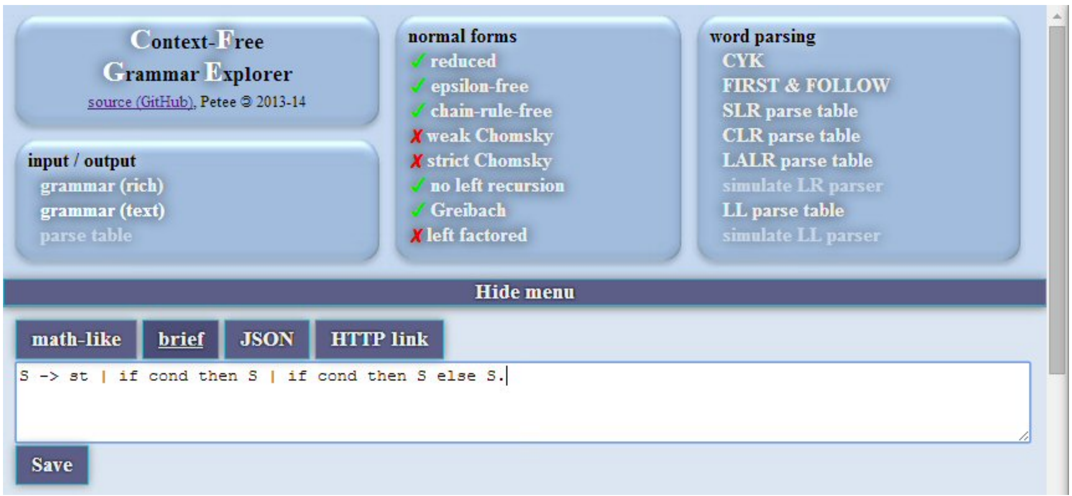

Figure 5.1: Entering a grammar using the brief text input method

This same grammar is also available as one of the built-in examples, located underneath the text input. The menu immediately shows us that the grammar is not left factored, which is one of the conditions necessary in order for a grammar to be LL. We therefore modify our grammar using the left factorization algorithm (**left factored** under normal forms is the menu).

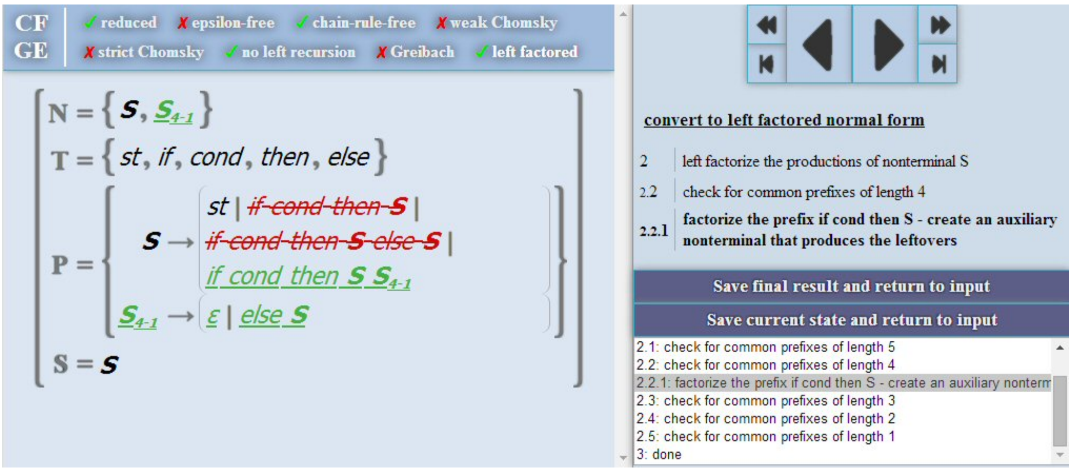

Figure 5.2: In the middle of visualizing the left factorization algorithm; notice the highlighted changes done in the current step

The algorithm visualization can be navigated using the six buttons at the top of the right panel, or using the list of steps at its bottom. We now save this modified grammar (**Save final result and return to input**); it will automatically be open in the rich input method. Since we are already there, we might want to change the name of the nonterminal S_{4-1} created by the conversion algorithm. We can do that by clicking on the name in the nonterminal list, entering a new name of our choosing and then releasing the focus on the text field by clicking elsewhere.

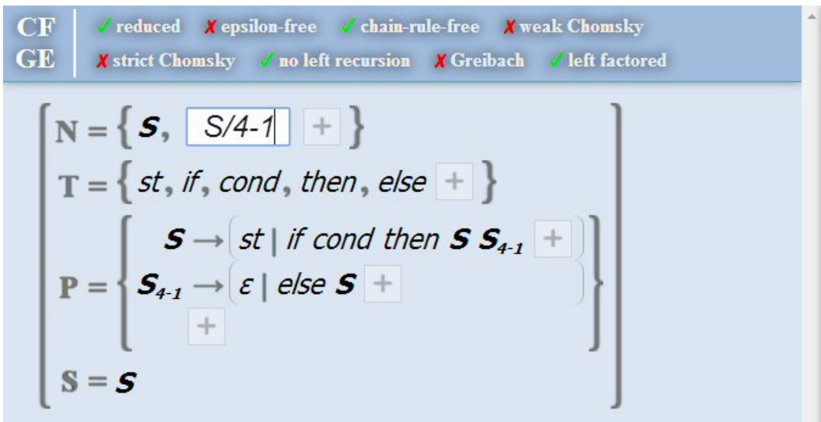

Figure 5.3: Using the rich input method to rename a nonterminal

We choose the name E, which will immediately replace the old name in all the grammar's productions. Now we go ahead and generate the **LL parse table**.

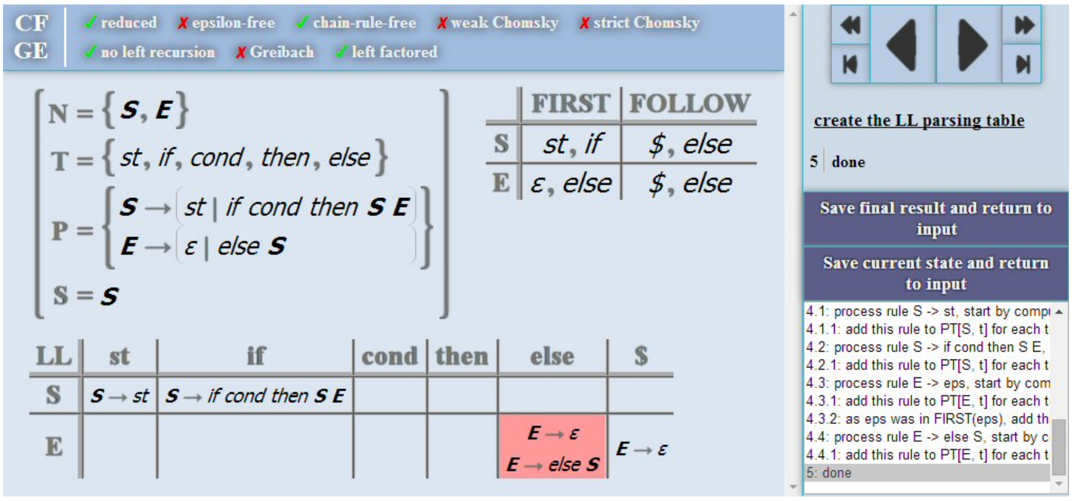

Figure 5.4: Finished visualization of the LL parse table generation algorithm; notice the highlighted conflict

We can see that the LL parse table has conflicts and thus even our modified grammar is not LL. This is unsurprising, as it is ambiguous. Let us choose to save the result, which gets us to the parse table editor. In some cases, disabling a parse table action can fix the conflict without changing the accepted language. In our case, this can be achieved by disabling the $E \to \varepsilon$ production in the cell with the conflict. This way the parser should associate any *else* with the most recent if that doesn't have such association already (this way the problem is resolved in many programming languages).

We will therefore click on this production in the parse table editor to disable it. Now, let us test this parse table by **simulating the LL parser**. The algorithm first prompts us to enter an input word; we do so and click **Parse**. The last step of the visualization shows us that the word was parser successfully. We can also the a visualization of the input word's parse tree.

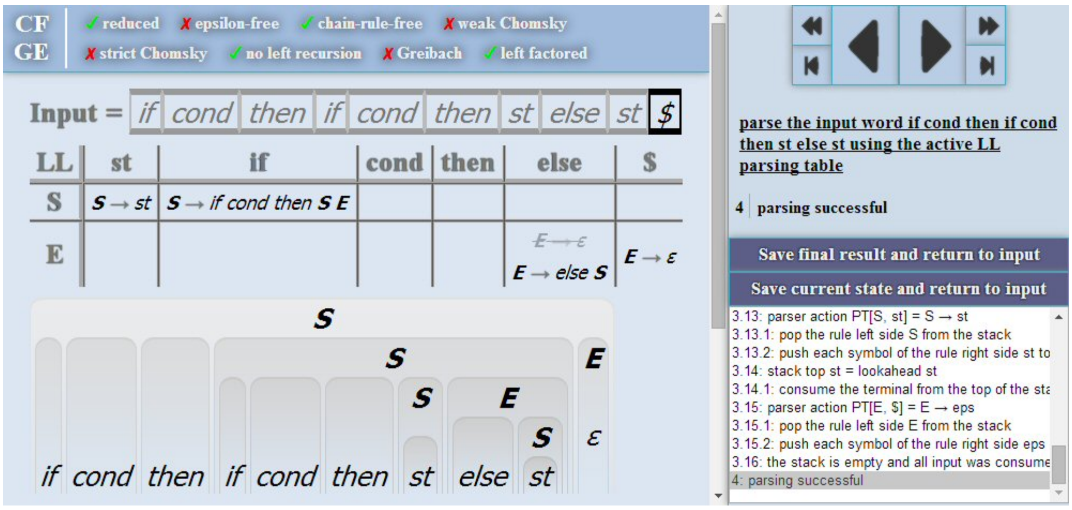

Figure 5.5: Finished visualization of a successful LL parser simulation; notice the parse tree

Chapter 6

Future plans

We would like to continue developing this application in the future, mainly to address any needs that may arise when the application is used as an aid in education. The following are some of the feature additions that are already planned.

- **Visualization of CYK**
  At the time of writing this thesis, the CYK algorithm is not implemented using the visualization framework created in this project. Due to this reason, step-by-step browsing of the algorithm's execution is not supported.

- **Computation of the $FIRST$ set for any input word**
  At the present time, the algorithm for computing $FIRST$ sets only does so for grammar nonterminals. In the future, this algorithm will allow the user to choose any input word to compute the $FIRST$ set for.

- **Translation support**
  The application was created with translation support in mind and with some more work, it will be possible to extend the application with languages other than English.

- **Better visualization of large structures**
  The application currently suffers on usability if the visualized structures (mainly parse tables and LR automata) are very large. In the future, we would like to address this by adding more controls to change their behavior.

Conclusion

The goal of this thesis was to create a tool to aid study in the field of syntax analysis or parsing. This should have been achieved by the means of a step-by-step visualization of several algorithms for context-free grammar transformation, parse table creation and parser simulation. The tool, a web application, should enable the user to freely choose the input for these algorithms, easily navigate in a hierarchical structure of the execution's steps and clearly visualize the structures used by the algorithm, distinguishing what was done in each step.

In order to achieve this goal we have firstly analyzed some of the existing tools in this area and the needs for a useful algorithm visualization. We have then created a library for representing various formal structures that are simple to use within the implementation of an algorithm and great effort was put into visualizing these structures clearly, mainly in the case of automata graphs. A number of input methods was created as well. We also designed a somewhat more general-purpose framework that allows to freely browse the execution of an algorithm and yet doesn't obscure the algorithm's implementation. Finally, altogether 15 algorithms were implemented using this framework, with brief descriptions for each step. In case of grammar normal forms, the tool also implements routines to check for a grammar's compliance.

The thesis explained the concepts behind the framework and discussed the interfaces provided by most classes, giving others the possibility to contribute to the project with even more algorithms in the future. A good example of use was provided for beginner users.

We believe this tool is an exceptional alternative to existing tools, mainly because it implements many algorithms, offers a high rate of flexibility in their visualization, has an appealing user interface and it is easily available to its potential users due to being a client-side web application in English. We therefore believe it will be a very useful teaching aid for courses about compilers, syntax analysis or formal languages in general and may have even greater potential if used for self-study of this important topic.

The application is hosted at: `http://micro.dcs.fmph.uniba.sk/cfge/`

Bibliography

[ALSU06] Alfred V. Aho et al. *Compilers: Principles, Techniques, & Tools, Second Edition*. Addison-Wesley, 2006.

[DaiGram] Michael Daines. *Grammophone*. 2013. URL: http://mdaines.github.io/grammophone/.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[MDNcEL] Mozilla Developer Network and individual contributors. *JavaScript Guide: Concurrency model and Event Loop*. 2013. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/EventLoop.

[MDNcOOP] Mozilla Developer Network and individual contributors. *JavaScript Guide: Introduction to Object-Oriented JavaScript*. 2014. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript.

[MM97] David Modjeska and Anna Marsh. *Structure and Memorability of Web Sites*. 1997. URL: https://www.ischool.utexas.edu/~i385e/readings/Modjeska.pdf.

[RF13] Branislav Rovan and Michal Forišek. *Formálne Jazyky a Automaty*. 2013. URL: http://foja.dcs.fmph.uniba.sk/materialy/skripta.pdf.

[RodJFLAP] Susan H. Rodger. *JFLAP*. 2009. URL: http://jflap.org/.

[UCCFGC] University of Calgary Department of Computer Science. *The Context Free Grammar Checker*. URL: http://smlweb.cpsc.ucalgary.ca/.

Appendix A

The application's source code

The thesis includes the source code of the application, attached as a CD in the printed version. We also recommend the reader to reach for the most recent version at:
`https://github.com/petee-d/cfge`