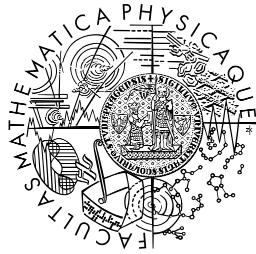


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Jan Veverka

Generator of Exercises on Automata and Grammars

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Martin Svoboda, Ph.D.
Study programme: Computer Science (B0613A140006)
Study branch: General Informatics Bc. (NIOI19B)

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date July 18, 2024 . . . .

Author's signature

I would like to express my sincere gratitude to my supervisor for his invaluable guidance and support throughout this project.

Special thanks to my grandma for providing a place to stay, which greatly facilitated my research and writing process. I am deeply grateful to my mom for her continuous encouragement and support. I also extend my thanks to my girlfriend and her mom for their understanding and assistance. Lastly, I would like to acknowledge my dad and my best friend for his ongoing support and encouragement.

Title: Generator of Exercises
on Automata and Grammars

Author: Jan Veverka

Department: Department of Software Engineering

Supervisor: RNDr. Martin Svoboda, Ph.D.

Abstract: This thesis introduces a Python-based tool designed to generate diverse and original problem assignments in formal languages theory, encompassing tasks such as determinization, the CYK problem, and conversion to a proper CFG. The tool allows the specification of desired properties for the generated problem assignments. Moreover, it allows us to facilitate the creation of variations of these assignments, ensuring their reuse across different exam dates. Each problem assignment includes a detailed report that covers the solving process, problem properties, and final solution, all formatted in LaTeX for easy integration into educational materials. Utilizing JSON configuration files, the tool offers flexibility and supports the seamless addition of new problems. By addressing challenges in the problem generation and solution validation, this tool serves as a valuable resource for enhancing teaching and assessment of formal languages theory.

Abstrakt: Tato práce představuje nástroj založený na Pythonu, který je navržen pro generování různorodých a originálních úkolů v teorii formálních jazyků, zahrnujících úkoly jako determinizace, CYK problém a převod na správnou bezkontextovou gramatiku. Nástroj umožňuje specifikaci požadovaných vlastností pro generované úkoly. Kromě toho usnadňuje tvorbu variant těchto úkolů, což zajišťuje jejich opakované použití při různých termínech zkoušek. Každý úkol obsahuje podrobnou zprávu, která zahrnuje proces řešení, vlastnosti problému a konečné řešení, vše formátované v LaTeXu pro snadnou integraci do vzdělávacích materiálů. Pomocí konfiguračních souborů JSON nástroj nabízí flexibilitu a podporuje bezproblémové přidávání nových problémů. Řešením výzev při generování úkolů a ověřování řešení slouží tento nástroj jako zdroj pro zlepšení výuky teorie formálních jazyků.

Keywords: Automaton, Grammar, Problem, Generator, Solver

Klíčová slova: Automat, Gramatika, Úloha, Generátor, Řešič

Contents

Introduction 4

1 Existing Solutions 5
- 1.1 Existing Generators . . . 5
- 1.2 Automata and Grammar Libraries . . . 8
  - 1.2.1 Pyformlang . . . 8
  - 1.2.2 JFLAP . . . 12
  - 1.2.3 AutomataLib . . . 13
  - 1.2.4 Algorithms Library Toolkit . . . 15
  - 1.2.5 Libraries Comparison . . . 17

2 Preliminaries 20
- 2.1 Basic Notions . . . 20
- 2.2 Grammars and Chomsky Hierarchy . . . 20
- 2.3 Automata . . . 23
- 2.4 Context-Free Languages . . . 27

3 Specification 30
- 3.1 Formal Language Framework . . . 30
  - 3.1.1 Grammar Module . . . 31
  - 3.1.2 Finite Automata Module . . . 31
  - 3.1.3 Formal Language Object . . . 32
- 3.2 Generic Requirements . . . 33
- 3.3 Main Program . . . 34
  - 3.3.1 Problem Class . . . 34
  - 3.3.2 Console Application . . . 39
  - 3.3.3 Non-Functional Requirements . . . 41

4 Problems 42
- 4.1 Determinization of NFA Problem . . . 42
  - 4.1.1 Algorithm . . . 42
  - 4.1.2 Properties . . . 43
  - 4.1.3 Generation Process . . . 44
- 4.2 CYK Problem . . . 45
  - 4.2.1 Algorithm . . . 45
  - 4.2.2 Properties . . . 48
  - 4.2.3 Generation Process . . . 49
- 4.3 Proper CFG Problem . . . 50
  - 4.3.1 Algorithm for Exclusion of Redundant Symbols . . . 50
  - 4.3.2 Algorithm for Exclusion of ε-rules . . . 52
  - 4.3.3 Exclusion of Simple Rules Algorithm . . . 52
  - 4.3.4 Proper CFG Algorithm . . . 53
  - 4.3.5 Properties . . . 54
  - 4.3.6 Generation Process . . . 56

5 Documentation **64**
5.1 High-Level Programmer Documentation 64
5.1.1 Formal Language Framework 64
5.1.2 Main Application 65
5.2 Testing 67
5.3 User Documentation 68
5.4 Instalation Documentation 72

Conclusion **74**

Bibliography **75**

List of Figures **77**

List of Tables **78**

A Attachments **79**
A.1 `generator_application.zip` 79

Introduction

Formal Languages theory is a fundamental area of computer science, with significant applications in compiler design, artificial intelligence, and formal verification. Central to this field are the study and manipulation of automata and grammars, which are essential for the understanding of language recognition and generation. Creating problem assignments for formal languages theory is crucial for evaluating understanding of a student and application of these concepts. However, generating original and diverse problem assignments that accurately test specific skills and situations can be challenging, especially when there are multiple exam dates and a need for variation.

The primary challenge addressed in this thesis is the difficulty in creating original and diverse problem assignments. Specifically, we aim to generate problems that test particular properties and situations within automata and grammar theory. Another key aspect is the necessity of providing solutions and the process of solving the generated problems to ensure their correctness and educational value.

The objective of this thesis is to develop an application capable of generating problem assignments based on configurable properties and situations. This application will not only generate diverse problem assignments but also create variations of these problems to reuse good problem assignments. Additionally, the application will provide a report with all generated assignments and their variations, solutions, and detailed solving processes for each generated problem assignment.

Another objective for the application is that it will support different problems and will be expandable with other problems.

The scope of the work includes the following types of problems: the determinization of NFA to DFA, the CYK problem, and the conversion to proper CFG problem.

Structure of the Thesis

This thesis is organized as follows:

- **Chapter 1: Existing Solutions** – We examine the existing solutions for generating problems and creating formal language frameworks necessary for this project.
- **Chapter 2: Preliminaries** – We define the essential concepts of automata, grammars, and formal languages.
- **Chapter 3: Specification** – We outline the requirements and specifications for the application and formal language framework.
- **Chapter 4: Problems** – We investigate the algorithms for solving specific problems, properties to be tested, and configuration for the problem generation.
- **Chapter 5: Documentation** – We provides high-level programmer documentation, user guides, and installation instructions for the application.

- **Chapter 6: Conclusion** – We summarize the findings, discusses the implications of the work, and suggests potential areas for future research.

By addressing these components, this thesis aims to contribute to the field of formal languages theory by providing a robust tool for generating and solving diverse problem assignments.

1. Existing Solutions

First, we explore the existing solutions for assignments generation for automata and grammars. This involves checking out various resources like websites, apps, and other online platforms. Unfortunately, we cannot find anything like existing generator for an automata and grammars problems but there are other resources related to it. Many of them already have predefined sets of assignments, while others offer tools to verify certain grammar rules. We are not limiting our search to automata and grammars; we are also curious about how other problem assignments, mainly math problems, are created. Since there are tools available for generating math problem assignments, we think there might be some techniques or ideas that we can borrow or learn from.

In the second part, we turn our attention to the technic side of things. We want to understand known libraries in different programming languages for automata and grammars and what they can do. By studying these libraries, we can get a better idea of what is possible and how to build our own framework.

In summary, the aim of this chapter is twofold: to understand what is already out there and to gather the necessary tools and knowledge to program basic framework, on which we will build.

1.1 Existing Generators

We will search for tools or methods related to generating problem assignments from automata, grammars, regular expressions, formal languages and topics from different subjects, or something what is related to formal languages.

There are numerous websites and online platforms that provide predefined sets of problem assignments. By *assignments* we mean all kind of assignments, multiple choice, open, or yes or no questions. Take, for instance, website Sanfourdy [1]. It includes a collection of over 1000 multiple choice questions spanning various formal language topics. However, there is the absence of any randomness. A few other websites introduce unpredictability by selecting a random subset of problem assignments from a larger pool. Even these predefined assignments can help us in inspiring of what kind of assignments we can create by our generator.

Another type of web applications we can find are validators that check if a grammar is context-free, regular, et cetera.

For example, there is a web application CFG Developer [2]. It was created by Christopher Wong at Stanford University in 2014. This web application offers the ability to create a grammar, automatically verify if the grammar is context-free, and if it is not, it will not generate examples of strings in this grammar. Similarly, if the grammar is context-free, it generates up to 7 examples of different strings that belong to the language of this grammar. In the beginning of creation of a grammar, there is always a rule from the starting symbol to the epsilon and we can not delete it, we can just change the right side of it. Then there is a button for adding a new rule. After we create a context-free grammar, we can test the acceptance of the set of strings, where it employs the Earley parser algorithm [3] for parsing the strings. There is the *Example* button, which generates an example of a context-free grammar. Unfortunately, it generates the same grammar every

time.

Up to now, we have discovered web applications and websites with pre-defined problem assignments on this topic. Our next step is to explore tools or methods related to the generation of math problem assignments from elementary to high school. As for subjects like Biology, Geography, and others, the problem assignments are typically created by humans. We might replicate or adopt similar methods to create our problem assignments. Some might employ artificial intelligence to devise these problem assignments, but that falls outside our primary focus.

Finding generators for math problem assignments is not challenging. Most of them are web applications and follow a similar principle. Consider the *Question Generator* from the web application MathsBot [4] as an example. It is a web application, where we can create three sets of problem assignments (1 to 25 problem assignments in each set). It covers topics from math up to high school. We can choose an interval of difficulty (from 1 to 10), which means setting borders of minimal difficulty and maximal difficulty of problem assignments. Of course, after generating problem assignments, we can show the answers by clicking on the problem assignment or the button *Show Answers*.

There is no explanation of how this generator works, but after creating a bunch of problem assignments, we observe that it is based on a randomness algorithm with templates. Most likely, creators write down problem assignment templates for each topic, and then there is a random generator with constraints that generates the constants for each problem assignment template. Then there has to be some function for parsing the constants into the template and a function for each template type, to calculate the correct output based on generated constants. The difficulty of the problem assignment could be determined by the constraints used for the random generator or by the template itself. For example, when creating a problem assignment for adding two numbers $x + y$, the difficulty depends on the constraints for the generator for constants. The difficulty will be lower when the constants are numbers from 0 to 9 versus when they are from 10 to 99.

Another interesting tool for generating math problem assignments is OneNote [5], which is a digital note-taking application developed by Microsoft. There we have an option to generate a quiz based on the equation we are solving. First of all, we have to solve an equation using Math Assistant, which is a special function in OneNote. After that, there will be an option titled *Generate a practice quiz*. We can choose to generate up to 20 similar problem assignments. This tool can generate equations with basic arithmetic operations on integers, such as addition, subtraction, division, and multiplication. It can also create linear or quadratic equations and inequalities, or many types of expressions with polynomials to factor, expand, integrate, or differentiate them. The generated problem assignments are multiple-choice, and Microsoft says that "distraction" answers (answers that are wrong but could be easily considered right) can take into account common mistakes that can be made while solving that specific problem, but they do not say how it works.

Continuing our search for math problem generators, we encounter platforms utilizing artificial intelligence, as anticipated, such as Questgen [6]. They cannot serve as an inspiration, but we can take inspiration from the following article.

The article *Algorithmically Generating Questions* [7], written by Jonathan Goldman and published in September 2013, is closely aligned with our topic. It outlines two strategies for generating problem assignments, particularly for math. These two approaches are solution-oriented and template-based.

The solution-oriented approach is about generating problem assignments based on the set of skills and concepts required to solve them. For instance, if we aim to craft an equation that requires adding two numbers and our goal is to test the ability of the student to carry a digit, we have to think of two numbers where this will happen and not only test the correctness of the result. In this context, the author mentions the publication by Erik Andersen, Sumit Gulwani, and Zoran Popović [8].

The publication is about analyzing and building a framework for abstracting the characteristics of a given procedure, in our case the procedure of addition. Jonathan Goldman mentions page three of his article. There is a pseudo-code of an algorithm for adding two integers, where the output of the algorithm is a trace. By the trace, we mean a string, where each character means some sort of skill or partial procedure, which the algorithm had to do to add the two integers. Let us consider that we have an algorithm for the addition of two integers, which imitates the procedure as we would do it on paper. We track properties of this problem by these characters: `A` stands for adding two digits, `C` stands for carrying, and `F` stands for final carrying. Final carrying is when we still carry a digit at the end of the addition but do not have anything else to add. We will create the traces for two addition assignments. For `1+1` we have the trace `A`, and for `112 + 993` we get the trace `AACAAF`. Based on the traces, we can say in which addition assignment we are going to test the skill of carrying a digit.

By generalizing this, we can execute the algorithm, monitor the trace, and categorize the task based on the tested skills and difficulty.

The second approach, discussed in the article, is template-based. We have already encountered this in the section on math problem generators. One of the examples could be to find all roots of $ax^2 + bx + c$. In this case, the quadratic equation is our template. Indexes a, b, and c are parameters to change, or we could say 'holes that must be filled'. We can restrict the template to have integer indexes from 1 to 10, then we have $10^3 = 1000$ possible problem assignments.

A big advantage of this approach is that it is easily generalizable and there is no problem in creating a huge amount of problem assignments. But on the other hand, we do not know how hard the problem assignment will be, or in which category we should put it. For example, the majority of the roots of the quadratic equation will not be integers. To generate problem assignments with this approach, we need to create some sort of validator, which will confirm that the generated values for a template lead to the desired type of problem assignment. Then we can randomly generate values and run them over the validator. There could also be the problem that only a small fraction of the generated problem assignments are the ones we are looking for. In this case, it could be desirable to involve some sort of heuristic for generating values, or maybe for quick recognition of good or bad problem assignments.

As far as we know, there are no existing generators of problem assignments for automata and grammars, but this article is a very useful insight into how to generate problem assignments in general and its thoughts will be definitely useful

in our future building of a generator.

1.2 Automata and Grammar Libraries

We will now focus on the existing libraries implemented in some of the most relevant programming languages because we will certainly need to represent automata and grammars within the framework for the generator we intend to propose.

1.2.1 Pyformlang

Pyformlang [9] is one of the most used frameworks for formal languages, automata and grammars for Python, written in Python. It was released as an open-source project by the main author Julien Romero on May 5, 2019. The library has continued to receive active maintenance and updates since its initial release. Its dedicated community of contributors and maintainers ensures that Pyformlang remains reliable, functional, and compatible with the latest Python versions or other dependencies. One part of the project is, of course, the official documentation. It is written with numerous examples of usage.

The library is segmented into eight distinct modules, each housing its unique set of classes. These modules are *Regular Expression*, *Finite Automaton*, *Finite State Transducer*, *Context Free Grammar*, *Push-Down Automata*, *Indexed Grammar*, *Recursive State Automata*, and *Feature Context Free Grammar*. Even though not all of the enumerated modules are relevant to our objective, we are interested at least in five of them.

Let us explore these modules to understand their composition and capabilities.

The *Regular Expression* module consists of three primary classes: `Regex`, `PythonRegex` and `MisformedRegexError`.

The `PythonRegex` class represents a regular expression as the built-in Python class `re` for regular expressions. A few additional features such as the positive closure were added.

The `Regex` class uses operators differently, for example, the union is represented either by | or `+`, while the `+` symbol denotes positive closure in the `PythonRegex` class. All of these operators could also be a part of the alphabet. Another surprising difference is that the alphabet in Pyformlang is not reduced to individual characters in terms of ordinary strings from the programming point of view. For instance, in Pyformlang, 'matfyz' is treated as a single symbol, whereas in Python, it is treated as a concatenation of six characters. We can see this situation bellow:

```
>>> r1 = Regex("matfyz|d")

>>> print(r1.accepts(["matfyz"]))
True

>>> r2 = PythonRegex("matfyz|d")

>>> print(r2.accepts(["matfyz"]))
False
>>> print(r2.accepts(["m", "a", "t", "f", "y", "z"]))
True
```

There are several classes in the *Finite Automaton* module, but main classes are `FiniteAutomaton`, `DeterministicFiniteAutomaton`, `NondeterministicFiniteAutomaton`, `EpsilonNFA`. The class `FiniteAutomaton` represents a general automaton and cannot be used directly. Other classes in this module inherit from it. The inheritance is in this order `FiniteAutomaton` → `EpsilonNFA` → `NondeterministicFiniteAutomaton` → `DeterministicFinite-Automaton`. Assuming that `A` → `B` means `B` *inherits from* `A`. We can see that the inheritance order corresponds to the order in which we transform one type of automaton to another when we try to do it "on paper". For example, if we want to convert a non-deterministic automaton to a deterministic, we can do it easily by using the function `to_deterministic()`. To be sure, we check if the automata, are equivalent to each other:

```
>>> nfa = NondeterministicFiniteAutomaton()
...
>>> dfa = nfa.to_deterministic()
>>> nfa.is_equivalent_to(dfa)
True
```

We have two options to define an automaton. The first one is to create an empty class object corresponding to the type of an automaton we will build. Afterward, we can freely add or remove states and transitions or declare which states are initial and final. The second one is to include all parameters properties while creating the object.

Beyond the core functionality of determining if a string is accepted by an automaton, the library offers various other functions for automaton objects. For example, `minimize()`, `reverse()`, `is_acyclic()`, `to_regex()`, or `from_networkx()`. All examples are self-explanatory, with the exception of the last one.

The last example is connected with the NetworkX [10] library. NetworkX is a Python library for studying, creating, and manipulating networks and graphs. The class `FiniteAutomaton` implements functions `from_networkx()`, `to_networkx()`, and `write_as_dot()`. The function `from_networkx()` will transform a graph of `networkx.MultiDiGraph` type to the corresponding automaton and `to_networkx()` will analogously transform an automaton to a graph. Note that the graph must adhere to a specific format. The `write_as_dot()` function is used for the visualization of an automaton. The outcome is the automaton in a DOT format written into the filename given as an attribute.

A basic DOT representation of an automaton consists of a digraph keyword followed by a set of statements enclosed in curly braces .

For example, an automaton with states A and B, and a transition from A to B on the symbol x, would be represented in DOT as:

```
digraph automaton {
    A -> B [label="x"];
}
```

Here is an example of creating an automaton, minimizing it, and then applying the `write_as_dot()` function, and final visualization
(see Figures 1.1 and 1.2):

```
dfa = DeterministicFiniteAutomaton()
dfa.add_transitions(
    [
        (0, "b", 1),
        (0, "a", 2),
        (1, "a", 1),
        (1, "b", 2),
        (2, "a", 2),
        (2, "b", 2),
    ]
)

dfa.add_start_state(0)
dfa.add_final_state(1)
dfa.add_final_state(2)

minimized_dfa = dfa.minimize()

dfa.write_as_dot(filename1)
minimized_dfa.write_as_dot(filename2)
```

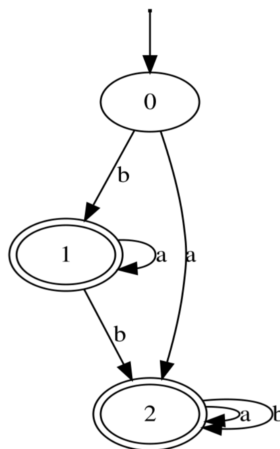


Figure 1.1: Visualization of DFA using Pyformlang

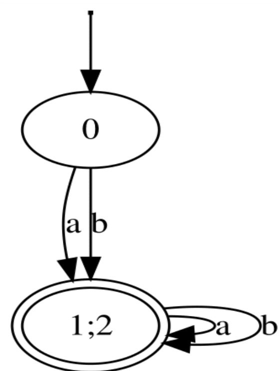


Figure 1.2: Visualization of minimized DFA using Pyformlang

As regular expression and finite automaton are equivalent, this library offers functions for converting between them.

The next module is the *Context Free Grammar* module, which has one main class, `CFG`, along with other smaller classes for terminals, non-terminals, etc. There exists a special class for an epsilon metasymbol named `Epsilon`. This representation removes ambiguities. Creating a `CFG` object is analogous to creating a `FiniteAutomaton` object in the *FiniteAutomaton* module. We define non-terminals (the official documentation and source code use the term *variables* instead of *non-terminals*), terminals, a start symbol, and production rules. The difference is that we can either create an empty automaton and then start adding states, transitions, and so on, or include it all, in parameters while creating the object in the *Finite Automaton* module. e have only the second option to include symbols and rules in the parameters while creating a `CFG` object here. Afterward, we can modify a grammar by a few functions such as `concatenate()` for concatenation of two grammars, `eliminate_unit_productions()` which return an equivalent context-free grammar without unit productions, `intersection()`, for example, with a regular expression or a finite automaton, etc.

We should mention non-editing functions such as `contains()`, `get_closure()`, `get_positive_closure()`, and `get_generating_symbols()`...

`CFG` is capable of checking if a grammar is in Chomsky normal form by calling `is_normal_form()` or getting equivalent CFG in Chomsky normal form by calling `to_normal_form()`.

It is worth noting that the library offers a more user-friendly representation of the grammar: non-terminals begin with a capital letter, and any other strings are considered terminals. Importantly, all strings must be space-free.

Our final module of interest is the *Push-Down Automata*. The hierarchy of classes is quite similar to the *Context Free Grammar* module. We have one main `PDA` class and a few smaller ones. And again we have a dedicated class for the epsilon metasymbol, different from other *Epsilon* classes in other modules. `PDA` has many functions in common with functions of finite automata in the *Finite Automaton* module. There are several extending functions for the proper functionality of a push-down automaton, like `set_start_stack_symbol()` and extended `add_transition()`. In this implementation, a push-down automaton can either accept by the final state or by an empty stack and there exist functions for conversion between them. We should also note that we can visualize the automaton by transferring it into the DOT file by `write_as_dot()`.

It is a well-established fact that context-free grammars and push-down automata are equivalent in expressive power. In this library, conversions between the two are supported. However, it is crucial to note that the conversion is only possible when the push-down automaton accepts by empty stack. This constraint is technical and is set by developers of the library because in formal language theory push-down automata which accept by empty stack and by final states are equivalent and convertable to each other.

From a usability perspective, this library is intuitive, user-friendly, and includes many features for automata and grammar. One downside is that creating an automaton by hand on paper or using a graphical user interface could be more intuitive than manually writing down each transition in the source code.

1.2.2 JFLAP

JFLAP [11] (Java Formal Languages and Automata Package) stands as one of the most notable frameworks for formal languages, automata, and grammars. This Java-based tool is a collection of graphical tools ideal for both learning and simulation. Susan H. Rodger and her team from Duke University developed and released JFLAP as an open-source project on April 12, 1998. Thanks to an active community of contributors, JFLAP has seen consistent updates, ensuring its compatibility with the latest versions of Java. Furthermore, there is extensive official documentation complete with hands-on examples. Those interested can also refer to the book *JFLAP An Interactive Formal Languages and Automata Package* [11] for more insights.

JFLAP covers topics such as Regular languages, Context-free languages, and Recursively Enumerable Languages, among others. We will highlight its capabilities within these topics.

Upon first encountering graphical interface of the JFLAP, we are presented with a main menu featuring options like *Finite Automaton*, *Mealy Machine*, *Moore Machine*, *Pushdown Automaton*, *Turing Machine*, *Multi-Tape Turing Machine*, *Turing Machine With Building Blocks*, *Grammar*, *L-System*, *Regular Expression*, *Regular Pumping Lemma*, *Context-Free Pumping Lemma*.

Selecting the *Finite Automaton* option allows us to design deterministic and non-deterministic automata, either with or without epsilon transitions. After designing, we can test automaton with various strings to check for acceptance. There is also the ability to visualize operation of the automaton step-by-step or view a complete path of its state transitions. Additionally, we can convert the automaton to its deterministic form, minimize it, or convert it into a regular grammar or regular expression. These conversions can be done instantly or in a step-by-step manner.

After clicking on the *Pushdown Automaton* button, we can choose from two options, create a pushdown automaton with multi-character input or with single-character input. Building a pushdown automaton is similar to creating a finite automaton; however, when adding a transition, we must specify the stack behavior. The core functionalities are also similar, we can track, and ask for acceptance of a single or set of strings. Also checking if there is any non-determinism. Lastly, we can convert it into a grammar while tracking specific steps, or combine it with another automaton.

For Turing machines, we can build a single-tape or a multi-tape Turing machine, up to 5 tapes. Then we can simulate a run on a string or set of strings, test non-determinism, combine it with another automaton, and convert it into an unrestricted grammar.

Selecting the *Grammar* option empowers us to craft distinct grammar rules. Once the rules are established, JFLAP discerns its classification, whether it is right or left linear, context-free, Greibach-normal, context-sensitive, or unrestricted. To validate if a string belongs to the language generated by a grammar, we might employ the *Brute Force Parse* feature. This method evaluates the string and constructs a parser tree, autonomously determining which rules are used to expand the nodes. Alternatively, with the *User Control Parse*, we specify in each phase which specific rule should unfold a node, provided that the rule is applicable. Transitions between different representations are streamlined: right-linear

grammars can convert into finite automata, and context-free grammars find their counterpart in push-down automata. Furthermore, JFLAP facilitates the transformation of a context-free grammar into Chomsky normal form. Again, there is the option to navigate through each transformation progressively, observing every nuance.

When selecting the Regular Expression button, we are presented with an empty space to input our regular expression. This functionality permits the construction of expressions utilizing symbols like parentheses `(`, `)`, the Kleene star represented by `*`, `+` for concatenation, and `!` signifying the empty string. Moreover, we can employ any characters from the alphanumeric characters, excluding the specified operators.

All objects created within JFLAP can be saved as a JFLAP file or exported as images in formats such as JPEG [12] or PNG [13].

Very interesting option from the main menu are either the *Regular Pumping Lemma* or *Context-Free Pumping Lemma*. By choosing them we are directed to a window that presents various language examples. From here, we can select the desired language to apply the lemma. For illustration, if we select regular languages, the process for context-free languages is similar. Upon making this selection, the *Explain* button appears. Clicking this button provides insight into whether the language is not regular, offering an explanation via the pumping lemma. Our main goal in this, however, is to find a valid division of a string that can undergo the pumping process. The initial step involves selecting an integer value m. The computer then picks a string longer than the value of m from the language of the grammar. Subsequently, our task is to divide the string into three distinct parts: x, y, and z. These substrings must meet the conditions stipulated by the pumping lemma. If a selected division does not comply with the conditions of the lemma, it flags the inconsistency. In cases where the string, based on our division, is unpumpable, we can try again.

An intriguing feature is the ability to reverse roles with the computer. In this mode, the computer takes the initiative by selecting the integer m. We then provide a string exceeding the length of m. The challenge of the computer is to discern a pumpable partition.

This is very interesting feature, which other mentioned libraries in this thesis do not have. Compared to Pyformlang, this library offers more functionalities and thanks to that the implementation of the JFLAP objects such as a finite automaton is more complex, distributed across several classes.

If we continue in comparing JFLAP and Pyformlang, we can say that there are significant differences between them in the simulation of the finite automaton or the conversions. While Pyformlang offers a more straightforward, function-call approach, JFLAP provides an in-depth, step-by-step method. This makes JFLAP an invaluable resource for those keen on understanding the workings of automata and grammars. On the other hand, for those aiming to integrate its logic into their projects, a study of source code of the JFLAP becomes essential.

1.2.3 AutomataLib

AutomataLib [14] is a versatile framework in Java including variety of automata types, but not for grammars and regular expressions. It is not as well-known

as Pyformlang, but it provides a robust platform for those looking to delve deeper into the realm of automata theory and graph theory. Currently, it covers generic transition systems, Deterministic Finite Automata, Mealy machines, and also more advanced structures like Visibly Pushdown Automata. AutomataLib was originally released in 2015 as an open-source project by Malte Isberner and Markus Frohme. The library has garnered a dedicated community over time, ensuring its regular updates and maintenance.

Apart from its primary focus on automata theory, AutomataLib demonstrates proficiency in tasks tied to graph structures. Given that automata can be viewed as directed graphs, the library facilitates graph-related operations. Notably, it enables users to traverse automata using methods similar to the classic depth-first and breadth-first searches, or find strongly connected components. Although it might not offer specialized shortest path algorithms, one can still find the most direct route to a state, typically through breadth-first search.

Another a pivotal aspect of AutomataLib is model checking. Ensuring that automata and state machines operate as expected. Within its suite of functionalities, AutomataLib offers features that support comprehensive testing and verification. For even deeper verifications, especially when we work with the domain of Linear Temporal Logic (LTL) model checking, AutomataLib demonstrates compatibility with LTSMin [15], which is a toolset for manipulating labeled transition systems and for model checking. By leveraging specific functions within AutomataLib, we can smoothly integrate with LTSMin, enabling users to export automata models and tap into advanced verification capabilities of the LTSMin.

After inspecting the diverse features of AutomataLib, ranging from its graph-related capabilities to its integration with LTSMin for advanced model checking, let us further explore the specific functionalities AutomataLib offers in the domain of automata theory and how they stand out.

We can build a deterministic automaton, a non-deterministic automaton, or convert the non-deterministic to the deterministic, ask if a string is accepted but there is no function to easily simulate it step-by-step and track the progress. We can edit the automaton, remove transitions, make an intersection, union, and difference operations with another automaton. We can compute minimalization of the deterministic automaton. We can also check equivalence of automata. AutomataLib provides other advanced functionalities over an automaton, but they are not important for our purposes.

AutomataLib does not provide built-in GUI, but the visualization of an automaton can be done by exporting it as a DOT file and visualizing it using another tool.

Related to finite automata, there are classes and many functions for building and operating over Moore and Mealy automata. Functions such as for deterministic and non-deterministic automata.

What about regular expressions? There are no classes for implementing or parsing regular expressions. This absence implies that it is not possible to create an automaton directly from a regular expression within this framework. Consequently, there are no functions provided for conversions between automata and regular expressions.

Let us dive into Context-free languages. There is not a direct way to create

a push-down automaton. There is only an interface called `OneSEVPA`, which is interface for implementing a 1-single entry visible push-down automaton, which is a visible push-down automaton of specific structures and semantics. Every visible push-down automaton can be viewed as a type of push-down automaton, which means that visible push-down automata accept a subclass of context-free languages.

As could be expected, there is no way to represent a context-free grammar, context-sensitive, or recursively enumerable language, or create an automaton that would recognize it.

AutomataLib is powerful in representing graph structures, which includes finite automata, including Mealy and Moore automata. It offers more algorithms over them than Pyformlang but for our purposes, we can not find many usable methods and classes for other parts of formal language theory.

1.2.4 Algorithms Library Toolkit

The ALT [16] (Algorithms Library Toolkit) is a library written in C++ and developed at the Faculty of Information Technology of the Czech Technical University in Prague, since 2019. Its primary function is to provide implementations for various automata, grammars, and other common structures, coupled with mathematical algorithms operating over them. It is an open-source project created by authors Jan Trávníček, Tomáš Pecka, and Štěpán Plachý. Its modular architecture emphasizes the independence of data types and algorithms, ensuring easy maintenance and extensibility. One of distinguishing features of ALT is its algorithm query language, *aql*. This language, executed within the *aql2* shell, allows users to manipulate data structures by mimicking the bash syntax. In other words, the library is meant to be used directly in the command line. Another option is to use the WebUI version. Where we can create automata, grammars, regular expressions, and graphs, and run algorithms on them as well. The WebUI application primarily utilizes mouse controls, supplemented by a set of keyboard shortcuts for enhanced usability. It should be noted that there is no official guide for the WebUI application as of now. The ALT project is actively maintained, continually enhancing its functionalities and updating its features.

The library is divided into many modules. For example, the *alib2data* module houses the main datatypes, including automata, grammars, and regular expressions, and the *alib2xml* module enables XML parsing and composing, etc.

A summary of what the library covers, in the context of formal languages, is, finite automata, regular grammars, and regular expressions cover the regular languages area. Representations for strings and various trees are provided. Pushdown automata, context-free grammars, and their variants cover context-free language areas. Finally, tree automata and tree regular expressions represent data structures related to tree languages.

We can create many types of automata that accept regular languages. The firs option is to create it in the command line with a specific format in the *aql* language, or the second option is to parse it from a XML file. Analogically, there is a function for converting the automaton to a XML file. If we are creating it in the command line, as in the Pyformlang library, we have to specify which automaton we are building, for example, deterministic, non-deterministic, epsilon

non-deterministic, and so on. While building, we have to define the alphabet of the automaton. After we have built the automaton, we can run it on the string and check its acceptance, get the string of indexes where the automaton passed a final state, and get the reached state after reading the whole string. There are functions for minimalization, exclusion of unreachable states, or we can determinize the non-deterministic automaton. For any two automata of the same type, we can compare them, if they are equivalent, make a union or concatenate. There are functions, which return an automaton in the DOT, LaTeX, or GasTex format, to an output stream. Afterward, we can use it to visualize the automaton.

The library includes a class for representing regular expressions, which employs well-established operators: `+` for alternation, `*` for iteration, and `()` to establish priority. The symbols `#E` and `#0` are dedicated representations for epsilon and the empty set. Furthermore, the library introduces a distinction between concatenated symbols and individual alphabet symbols. An integral aspect is the space delimiter, where *11* represents a single symbol, *1 1* represents two distinct concatenated symbols. After we build a regular expression, we can convert it to the automaton, or an automaton to the regular expression, thanks to different algorithms, such as the Brzozowski's derivation algorithm [17], Glushkov's construction algorithm [18], or Thompson's Construction Algorithm [19]. We can also convert a regular expression to the corresponding regular grammar, which delegates to the Glushkov's construction algorithm, or to the corresponding right regular grammar, with Brzozowski's derivation algorithm. There is also the function for optimizing a regular expression, which means that it tries to transform the expression to be smaller but equivalent.

To accept a context-free language, we can build a non-deterministic push-down automaton or a less powerful deterministic push-down automaton. Let us note that the official documentation does not provide instructions on how to build it, whether in the command line or the web application. But we can read from the API documentation, that there are many similar functions as in the automata for regular languages. For example, in speaking of editing functions over the automaton, such as changing initial states, and adding, or removing transitions, the functions of non-deterministic, or deterministic automata are the same. Of course, we can run it over a string. We can convert it to the same format for the visualization, or to the XML file. But there are no functions for minimalization or union. We also cannot convert between push-down automata and context-free grammars.

For creating grammar, firstly, we have to specify the type of the grammar, `RIGHT_RG`, `LEFT_RG`, `CFG`, `GNF` (stands for context-free grammar in the Greibach normal form), `CNF` (stands for a context-free grammar in the Chomsky normal form), etc. After that, we include the set of non-terminals, the set of terminals, the set of rules, and the initial non-terminal symbol. Naturally, acceptance of a string is included. Additionally, there are various other functions, such as querying if the language generates epsilon and retrieving reachable symbols of the grammar. For two regular or context-free grammars, we can alternate them, or concatenate them. Regarding conversion, for a regular grammar, we can convert it to the finite automaton, to the left or right form, and afterward, the left regular, or the right regular grammar convert to the regular expression. We can convert regular and context-free grammars to Chomsky's normal form, or to

Greibach's normal form. Let us note that we can also create a noncontracting and an unrestricted grammar.

We can find the class for representation of a deterministic single tape Turing machine, which accepts recursive languages, in the API documentation. Unfortunately, as for the push-down automata, there is no explanation of how to create them with the command line interface, and there is no option to create them in the WebUI.

The Algorithms Library Toolkit excels when dealing with regular languages. However, for context-free languages and beyond, it lacks explanation on how to construct corresponding objects.

1.2.5 Libraries Comparison

For our summary of libraries, we will compare them in terms of usability, features, and functionalities. This will give us a better overview of each library, and we can easily navigate through them in the future, while building our framework for automata, grammars, regular expressions, etc.

First of all, we will compare them in a general context 1.1. Pyformlang and AutomataLib are both frameworks that can be imported into the code and used for their functionalities in our project. Conversely, JFLAP and ALT can be used on their own only with their graphical interface, while JFLAP is the primary graphical educational tool, the ALT library is designed to be used with query language *aql*, directly in the command line or written down in scripts that can be executed. Theoretically, we could import classes, and the whole logic of JFLAP and ALT into our code, in JFLAP case to our Java code and in ALT case to our C++ code, and use it as a framework for our project. This option could be quite tricky and would include studying their source code.

Library	**Written in language**	**Framework**[a]	**GUI**[b]
Pyformlang	Python	YES	NO
JFLAP	Java	NO	YES
AutomataLib	Java	YES	YES
ALT	C++	NO	YES

[a] Its logic can be integrated easily into our own project.

[b] Graphical User Interface

Table 1.1: General information about libraries.

The libraries are in active maintenance and are open-source. The oldest one is JFLAP, released in 1998. All of the others are quite new, released in 2015–2019 1.2.

All of the libraries can represent finite automata. The only library that cannot represent a regular expression and a push-down automata, and a context-free grammar is AutomataLib. Note that in ALT there exists a class for representing a push-down automaton but there is no explanation of how to create it. The only library that can represent and work with Turing machines is JFLAP.

What libraries can do with objects in regular languages context?

Library	Open-source	Released in	Active maintenance
Pyformlang	YES	2019	YES
JFLAP	YES	1998	YES
AutomataLib	YES	2015	YES
ALT	YES	2019	YES

Table 1.2: External information about libraries.

All of the libraries can represent and offer basic functionalities over finite automata, such as the acceptance of a string, converting the non-deterministic to a deterministic one, minimizing a deterministic automaton, etc. Also, all libraries have a function to somehow visualize the automaton. Either exporting the DOT file or in its own GUI.

The only library that cannot represent a regular expression is AutomataLib and it implies that we cannot convert between finite automata and regular expressions. But all other libraries have an option to convert between them

As we can see in a table 1.3.

Library	Basic[a] functionalities	Minimize DFA	Convert[b]	Visualize automaton
Pyformlang	YES	YES	YES	YES
JFLAP	YES	YES	YES	YES
AutomataLib	YES	YES	NO	YES
ALT	YES	YES	YES	YES

[a] Acceptance of a string, or editing automata.

[b] Conversion between a regular expression and a finite automaton, in both ways.

Table 1.3: Capabilities of libraries in regular languages.

A push-down automaton can be represented in the Pyformlang and JFLAP, despite there exists the class for a push-down automaton in ALT, we do not count it because we do not know how to use it. Both, Pyformlang and JFLAP, offer similar basic functionalities over them for finite automata, including visualization of them.

In the speaking of only context-free grammars context, we can represent them in Pyformlang, JFLAP, or ALT but only for ALT and JFLAP we can create another type of grammar. There is one big difference, in ALT we have to specify the type of a grammar, and in JFLAP we can write down rules and after that check the type of it. But let us focus now only on context-free grammars. The basic functionality, if a string is contained in a language the grammar generates, is included in all three libraries. Even the transformation to the Chomsky normal form is in all of them. Other functionalities are very similar in Pyformlang and ALT, for example, the concatenation of two grammars. Despite there being a bunch of options to convert between different types of grammar in JFLAP, other functionalities, such as concatenation, miss. The conversion between a push-down automaton and a context-free grammar is possible in JFLAP and Pyformlang. These properties are compared in a table 1.4

Library	CFG[a]	Other Grammars[b]	To Chomsky normal form	PDA[c]	Convert[d]
Pyformlang	YES	NO	YES	YES	YES
JFLAP	YES	YES	YES	YES	YES
AutomataLib	NO	NO	NO	NO	NO
ALT	YES	YES	YES	NO	NO

[a] Context-free grammar.

[b] Grammars of type 1 and 0.

[c] Push-down automaton.

[d] Conversion between a push-down automaton and a context-free grammar.

Table 1.4: Capabilities of libraries in context-free languages.

Only JFLAP offers the ability to follow conversions, testing acceptances, and other functionalities step-by-step. This feature is really powerful for learning and understanding how the algorithms in formal languages work. It is also the only library where we can build Turing machines, and even do something with the Pumping lemma. These reasons are, why it is very powerful. On the other hand, each library has something exclusive, that others do not have. For example, in which programming language we can use it, or how we can use it. In terms of functionalities, ALT is quite powerful, but the learning of the aql language could turn many people off. The Pyformlang is on the other hand straightforward in terms of usage, with well-written official documentation. For AutomataLib, we do not have many functionalities in the whole context of formal languages but it is strong in graph algorithms over finite automata and other graph structures, or in testing them.

2. Preliminaries

In this chapter, we will establish and introduce some concepts and objects from formal languages theory. We will define a *deterministic*, and an *non-deterministic automaton*, *grammar*, or a *regular expression* and then talk about how they work.

Upon theoretical basis, we will be building our framework for automata and grammars, analyzing algorithms for problems in our scope, and creating our intended generator.

2.1 Basic Notions

Let us start with the *strings*. Firstly, we will need an *alphabet*, so that we have something to build strings from. An *alphabet* is a finite set of symbols, usually denoted by Σ. For example, the binary alphabet is the set $\{0, 1\}$. A *string over an alphabet* Σ is a finite sequence of symbols from Σ. *Empty string* will be denoted by ε. Naturally, the *length of a string* x, denoted by $|x|$, is a number of symbols in the string. For an alphabet Σ, we define Σ^+ as the set of all non-empty strings over Σ and Σ^* is defined as $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$, which means it is the set of all strings over Σ.

We can do a few operations over strings. The operation of the *concatenation* of strings x and y is written as $x \cdot y$, or a shorter version xy. Another one is a *reversal of a string*. For a string $x = a_1a_2...a_n$, $x^R = a_na_{n-1}...a_2a_1$ is the reversal of the string x.

A *formal language* L over an alphabet Σ is a subset of Σ^*. There are basic set operations, such as *union*, *difference*, or *intersection*. Non-trivial operations are the concatenation of two languages or the n-th power of language and kleene star. The *concatenation* of two languages L_1 over Σ_1 and L_2 over Σ_2 is $L = L_1.L_2 = L_1L_2 = \{x.y : x \in L_1, y \in L_2\}$ and L is defined over the alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$. For $n \in \mathbb{N}$ is the *n-th power of a language* L derived from the concatenation as $L^n = L.L^{n-1}$ and $L^0 = \{\varepsilon\}$. The last mentioned operation *kleene star* L^* *of language* L is defined as $L^* = \bigcup_{n=0}^{\infty} L^n$.

The first thought about how to represent a language is that we can just list its elements. However, this approach becomes impractical when dealing with languages that have an infinite number of strings, in other words, language is not finite. Formally, a language $L \subset \Sigma^*$ is *finite* if $\exists\ n \in \mathbb{N}^0 : |L| \leq n$.

As we established fundamentals of formal languages, we can proceed to models that describe them.

2.2 Grammars and Chomsky Hierarchy

The solution, how to represent any language, even non-finite, lies in the use of grammars. A *grammar* serves as a formal specification for a language.

Definition 1 (Grammar)**.** *A **grammar** is a tuple $G = (N, \Sigma, P, S)$, where:*

- **N** *is a finite set of **non-terminal symbols**.*
- **Σ** *is a finite set of **terminal symbols**.* $\Sigma \cap N = \emptyset$.

- **P** *is a finite set of* ***production rules****. It is a subset of* $(N \cup \Sigma)^*.N.(N \cup \Sigma)^* \times (N \cup \Sigma)^*$*. An element* (α, β) *of* P *is written as* $\alpha \to \beta$ *and is called rule.*

- $\mathbf{S} \in N$ *is the* ***start symbol****.*

□

Non-terminal symbols represent syntactic variables in the grammar and are used to generate strings, but they do not appear in the final generated string. *Terminal symbols*, on the other hand, are the symbols from which the strings are built and cannot be further derived.

A *derivation* of a string in a grammar $G = (N, \Sigma, P, S)$ is a sequence of application of production rules. A *sentential form* represents a string in the process of derivation, serving as an intermediate step. A *sentence* β is special case of *sentential form* consisting solely of terminal symbols ($\beta \in \Sigma^*$), representing the final string derived.

The *generation* process begins with the start symbol S, which serves as the initial sentential form. Then we non-deterministically derive by applying production rules until we get a sentence, which is the final string we have generated and we say that *the string is generated by the grammar*.

By *applying a production rule*, we mean that we take a sentential form and find a substring that is located on the left-hand side of a production rule and then we replace that substring with the right-hand side of that rule, in other words, if we have a sentential form uXv, where and we apply a production rule $X \to Y$, we get a new sentential form uYv. Note that there is always at least one non-terminal symbol on the left-hand side of each production rule.

The derivation of uYv from uXv is denoted by $uXv \Rightarrow uYv$. If the derivation from a string α to a string ω is sequence of applying k rules we denote it as $\alpha \Rightarrow^k \omega$. By $\alpha \Rightarrow^+ \omega$, we denote $\alpha \Rightarrow^i \omega$, where $i \geq 1$, we derive in nonzero number of steps. And by $\alpha \Rightarrow^* \omega$, we denote $\alpha \Rightarrow^j \omega$, where $j \geq 0$, we derive in any number of steps, because we can always derive ω from ω by applying zero rules, $\omega \Rightarrow^0 \omega$.

There is an assumption that non-terminal symbols N and the terminals Σ are disjoint sets and that is essential. Because we could not distinguish between them if we use them in one sentential form. Then we would not know if the sentential form is sentence containing only terminal symbols an consequently, we cannot determine whether the generating process has terminated or should continue.

A *language* L *generated by a grammar* G is noted as $L(G)$ and it is a set of all sentences generated by the grammar.

Two grammars G_1 and G_2 are considered equivalent if and only if they generate the same language, denoted as $L(G_1) = L(G_2)$.

Let us demonstrate how to represent a language by a grammar and the derivation process. We have a language L over the alphabet $\{\mathtt{0}, \mathtt{1}\}$, where all strings begin with `0`, then have any number of `1`s and end with `0` again. This is impossible to represent it by writing down all of the strings but we can represent it by the following grammar G_1.

Example (Grammar).

$$G_1 = (\{\mathtt{S}, \mathtt{A}\}, \{\mathtt{0}, \mathtt{1}\}, P, \mathtt{S})$$

where P includes the following production rules:

1. $\texttt{S} \to \texttt{0A}$
2. $\texttt{A} \to \texttt{0}$
3. $\texttt{A} \to \texttt{1A}$

In this grammar, $\texttt{S}$ is the start symbol, and the terminal symbols are $\texttt{0}$ and $\texttt{1}$. Using this grammar, we can derive a sentence such as ($\alpha \Rightarrow_k \omega$ means, we derive ω from α by applying the rule number k):

$$\texttt{S} \Rightarrow_1 \texttt{0A} \Rightarrow_3 \texttt{01A} \Rightarrow_3 \texttt{011A} \Rightarrow_2 \texttt{0110}$$

□

Now, we introduce a hierarchy in grammars and the class of language that every type of grammar generates.

Definition 2 (Hierarchy of Grammars)**.** *Let $G = (N, \Sigma, P, S)$ be a grammar, then G s:*

- ***unrestricted*** *(type 0), if it satisfies a general grammar definition.*
- ***context-sensitive*** *(type 1), if every rule from P is of the form $\gamma A \delta \to \gamma \alpha \delta$, where $\gamma, \delta \in (N \cup \Sigma)^*, \alpha \in (N \cup \Sigma)^+, A \in N$, or the form $S \to \varepsilon$ if S is not present in the right-hand side of any rule.*
- ***context-free*** *(type 2), if every rule is of the form $A \to \beta$, where $A \in N$, $\beta \in (N \cup \Sigma)^*$.*
- ***regular*** *(type 3), if every rule is of the form $A \to aB$ or $A \to a$, where $A, B \in N, a \in \Sigma$, or the form $S \to \varepsilon$ if S is not present in the right-hand side of any rule.*

□

As we can see, with each lower class (higher type number) the conditions for the grammar rules are more strict, which leads to lower expressive power and the languages generated by these grammars are less complex.

Now let us see a corresponding hierarchy of languages.

Definition 3 (Hierarchy of Languages)**.** *The language L is:*

- ***recursively enumerable*** *(type 0), if there exists an unrestricted grammar which generates it.*
- ***context-sensitive*** *(type 1), if there exists a context-sensitive grammar which generates it.*
- ***context-free*** *(type 2), if there exists a context-free grammar which generates it*
- ***regular*** *(type 3), if there exists a regular grammar which generates it.*

□

It is called the Chomsky hierarchy.

2.3 Automata

Automata are models that take a *string* over the alphabet of an automaton and answer if the string is accepted or not. In this way, we can describe a language. We can build an automaton and the corresponding language will be composed of all strings accepted by that automaton. *Regular* languages (type 3) can *be recognized* by *finite-state automata*, in other words, for each regular language there exists a finite-state automaton which accepts all and only strings from the language.

We have a deterministic finite automaton or a non-deterministic finite automaton. They are equivalent in computing power, which means both of them recognize regular languages and each non-deterministic automaton can be converted to a deterministic one.

Definition 4 (Deterministic Finite Automaton)**.** *A* ***deterministic finite automaton*** *(DFA) is a tuple* $M = (Q, \Sigma, \delta, q_0, F)$*, where:*

- Q *is a finite* ***set of states****.*
- Σ *is a finite alphabet (****input alphabet****).*
- $\delta : Q \times \Sigma \to Q$ *is a* ***transition function****.*
- $q_0 \in Q$ *is the* ***initial state****.*
- $F \subseteq Q$ *is a* ***set of final (accepting) states****.*

□

The DFA processes an input string by starting in the *initial state* q_0 and, for each symbol in the input string (starting with the first symbol of the input string) it makes one *step*. By *step* we mean that the automaton transition into the new current state, based on the current state and next input symbol from the input string. To which state we will transition is determined by the transition function δ. *The input string is accepted* if and only if, after reading the last symbol, the automaton is in one of the final states in F. The input is rejected if we end in a state which is not final or the transition for the next symbol is not defined.

We can define an automaton by defining main elements and δ function, or by a diagram.

Let us build a simple DFA D_1, which will be over the alphabet $\Sigma = \{\texttt{a, b}\}$ and will accept only strings with even number of `a`.

Example (Deterministic Finite Automaton).

$$D_1 = (\{s_1, s_2\}, \{\texttt{a}, \texttt{b}\}, \delta, s_1, \{s_1\})$$

Where δ is defined as:

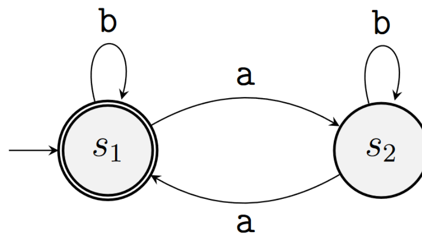
	a	b
s_1	s_2	s_1
s_2	s_1	s_2

The shorter option for defining the whole automaton is the table, which corresponds to the δ function. Initial state is denoted by an arrow pointing into it and final state by an arrow pointing out of it.

<->

	a	b
s_1	s_2	s_1
s_2	s_1	s_2

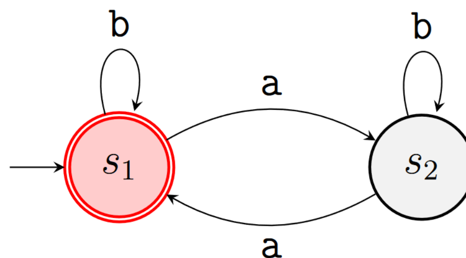
And lastly, a diagram, where nodes represent states. The node with an arrow pointing into it is the initial state and states with double bordering are final states. The input alphabet is implicitly defined by all symbols found on edges between nodes and these edges define the δ function.



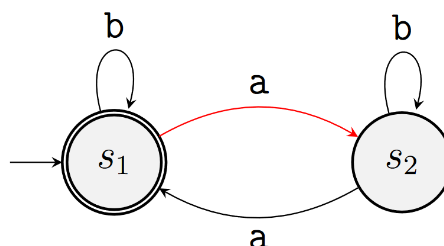
□

As we can see, a table and a diagram representation are equivalently strong to the plain definition but a diagram is the most intuitive way for explaining how an automaton accept or reject an input string.

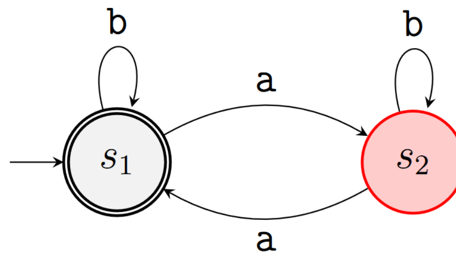
Example (Acceptance of an inptut string by a DFA). Let us have an automaton D_1 and an input string `aabab`. We will start in the initial state s_1.



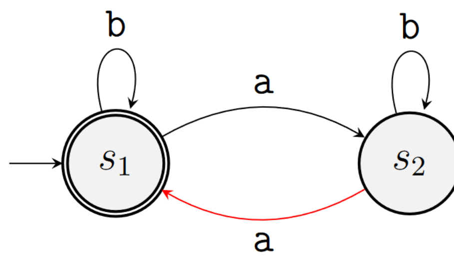
We are in the state s_1 and reading the first symbol in the input string `aabab` (first `a` underlined). Which means that we will transition through the edge with `a`.

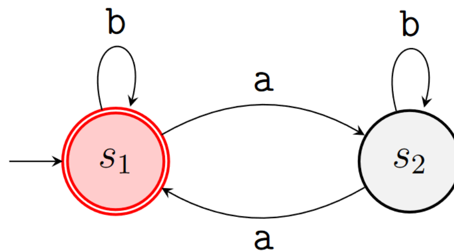


And the new current state is s_2.

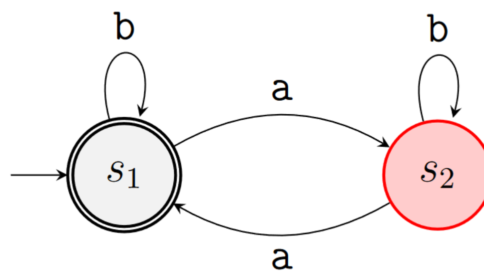


We will read the next symbol in the input string $\mathtt{a\underline{a}bab}$, which is `a` again, and we will transition to the state s_1.





We will iterate, until we reach the end of the input string and in this particular case we will end in the state s_2.



Because we have read the whole input string and we are not in the final state at the end, it implies that the input string `aabab` is *not accepted*. This corresponds to the fact that there are three occurrences of `a`, which is not an even number.

□

Equally powerful to a DFA is *a non-deterministic finite automaton*. The only difference is in the transition function.

Definition 5 (Non-deterministic Finite Automaton)**.** *A* ***non-deterministic finite automaton*** *(NFA) is a tuple* $N = (Q, \Sigma, \delta, q_0, F)$*, where:*

- Q *is a finite* ***set of states****.*
- Σ *is a finite alphabet (****input symbols****).*
- $\delta : Q \times \Sigma \to 2^Q$ *is a* ***transition function****, where* 2^Q *is a set of all subsets of* Q*.*
- $q_0 \in Q$ *is the* ***initial state****.*
- $F \subseteq Q$ *is a* ***set of final (accepting) states****.*

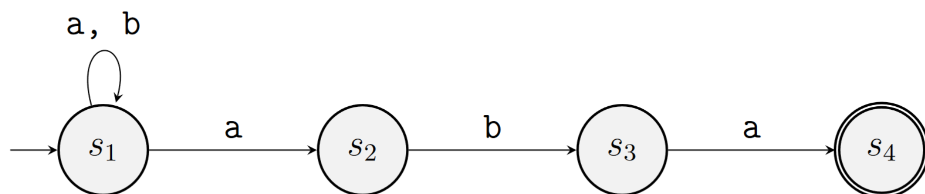
□

The NFA processes an input string, similar to a DFA, by starting in the initial state q_0, and for each input symbol, it takes one step. However, unlike a DFA, an NFA makes a non-deterministic transition in each step. Specifically, it non-deterministically (randomly) selects a new state from a set of possible states based on the transition function δ.

To clarify, at each step, given the current state and the input symbol, the NFA explores multiple potential paths by considering all possible next states from the set defined by δ.

The *NFA accepts the input string* if and only if there exists at least one sequence of steps such that, after reading the last symbol of the input string, it ends in one of the final states from the set F; otherwise, it rejects the input.

Example (Non-deterministic Automaton). We can create a NFA N_1 over an alphabet {`a`,`b`} which accepts only strings ending with `aba`. The automaton will look like this:



□

Here, we can see that it starts in the state s_1 and by processing the input string `aba`, it can end in states $\{s_1, s_2, s_4\}$. Therefore, it accepts this string because at least one of the ending states is final, specifically the state s_4.

The last two notes for automata are that two automata are equivalent if they accept the same language.There is a theorem that for each NFA there exists an equivalent DFA.

We introduced two types of finite-state automata and how to represent them and we know they are equally powerful in describing or defining regular languages as regular grammars. Let us mention that there are algorithms for conversions between regular grammars and finite automata.

2.4 Context-Free Languages

Languages of the type 2 in Chomsky hierarchy are generated by *context-free grammars*. They have smaller restrictions than regular languages and thus they are more complex. Rules of context-free grammars can be only in a form $A \to \beta$, where A is one non-terminal and β is any string build from terminals and non-terminals.

The generated language by a context-free grammar can be empty (it does not generate any string). For deciding if the langauge is empty we will have to know if the starting symbol S is a *generating symbol*.

Definition 6 (Generating Symbol)**.** *Symbol $A \in N$ in a context-free grammar $G = (N, \Sigma, P, S)$ is* ***generating*** *if there exists at least one derivation $A \Rightarrow^+ \omega$, such that $\omega \in \Sigma^*$, in other words, there exists at least one sentence that can be derived from the symbol A.*

□

If the starting symbol S is *non-generating*, we cannot generate any string from it and the language of the corresponding grammar is empty.

Another types of not useful symbols are *unreachable* symbols. They can be generating but we cannot reach them from the starting symbol.

Definition 7 (Reachable Symbol)**.** *Symbol $B \in (N \cup \Sigma)$ in a context-free grammar $G = (N, \Sigma, P, S)$ is* ***reachable*** *if there exists a derivation $S \Rightarrow^* \omega B \gamma$ where $\omega, \gamma \in (N \cup \Sigma)^*$, in other words, there exists at least one sentential form or sentence that includes B and can be derived from S.*

□

If we combine both of them then we have an unreachable or a non-generating symbol, we say it is redundant.

Definition 8 (Redundant Symbol)**.** *Symbol $X \in (N \cup \Sigma)$ is* ***redundant*** *in $G = (N, \Sigma, P, S)$, if there does not exist a derivation $S \Rightarrow^* \omega X \gamma \Rightarrow^* \omega x \gamma$, where $\omega, x, \gamma \in \Sigma^*$.*

□

Redundant symbols can be removed from a context-free grammar without loss of power because the grammar without redundant symbols still generates the same language.

The special type of a rule is an *ε-rule*. It is a rule where ε is on the right hand side (note that ε cannot be on the left hand side, it would not make any sence). We can have a special demand that a grammar will be *ε-rule free*.

Definition 9 (ε-rule Free Grammar)**.** *A context-free grammar $G = (N, \Sigma, P, S)$ is* ***ε-rule free****, if*

1. *P contains no ε-rule, or*

2. *P contains only one ε-rule of form $S \to \varepsilon$ and S does not appear on the right hand side of any rule in P.*

□

If we have a context-free grammar which can generate an ε, we can not delete all the ε-rules because we would no longer be able to generate it and the grammars would not be equivalent. Because of that there is the second option in the definition of ε-rule free grammar.

Another additional requirement for a context-free grammar could be that it must be *cycle-free*.

Definition 10 (Cycle-Free Grammar)**.** *A context-free grammar $G = (N, \Sigma, P, S)$ is* ***cycle-free****, if no derivation $A \Rightarrow^{+} A$ is possible for any $A \in N$.*

□

After combining all of these requirements, we get a special type of a context-free grammar, which is a *proper context-free grammar*.

Definition 11 (Proper Context-Free Grammar)**.** *A context-free grammar $G = (N, \Sigma, P, S)$ is* ***proper****, if it is cycle-free, ε-rule free, and without redundant symbols.*

□

We can convert every context-free grammar into a proper one, as we will explore further in our reading.

The last special rule is a *simple rule*. It is a rule of type $A \rightarrow B$, where A, B are both non-terminals. It can be even same non-terminals, $A = B$.

Theorem 1. *If a context-free grammar $G = (N, \Sigma, P, S)$ has no ε-rules and no simple rules, then it is* ***cycle-free****.*

□

To convert a context-free grammar into a proper context-free grammar, we need to eliminate ε-rules, followed by the removal of simple rules and redundant symbols. This process ensures that the grammar fulfills the definition of a proper context-free grammar.

As we try to restrict the production rules in a grammar, we can also put some restrictions on their format but with the same expression power. For example, for context-free grammars, we can define a grammar with a specific format of rules which is called Chomsky normal form.

Definition 12 (Chomsky Normal Form)**.** *Let be $G = (N, \Sigma, P, S)$ a context-free grammar, it is in Chomsky normal form (CNF) if all production rules are of a form:*

1. *$A \rightarrow BC$ (unit productions), where $A, B, C \in N$.*
2. *$A \rightarrow a$ (terminal productions), where $A \in N$, and $a \in \Sigma$.*

□

In CNF, there are no ε-rules instead of a rule of a form $S \rightarrow \varepsilon$, where the starting symbol is deriving ε and also there are no unreachable symbols.

In this chapter, we laid the foundational concepts of formal languages theory. We defined essential elements such as alphabets, strings, and formal languages. We introduced grammars as formal language generators, detailing their structure

and derivation process. The Chomsky hierarchy categorized grammars based on their expressive power, correlating with the types of languages they can generate. Finally, we introduced finite automata as models for recognizing regular languages, distinguishing between deterministic and non-deterministic variants. These concepts establish the groundwork for our subsequent discussions on automata and grammars theory.

3. Specification

In this chapter, we outline the detailed specifications of our project, which involves designing a framework for formal languages and developing a program to generate and solve problem assignments from formal language theory. We will analyze the structure of the framework, focusing on its modules and classes for automata and grammars, and specify the requirements and functionalities of the main program. This will include an in-depth look at how the program will be structured, generating problem assignments, decorating them, creating reports, and formating them for easy integration into exam papers.

3.1 Formal Language Framework

First of all, we will need a framework for formal languages. We would like to scope problems from finite automata theory and grammar theory. We could use Pyformlang [9], but we already know that there are many extra methods and utilities that we will not need. For example, algorithms over a context-free grammar, and there could be others that we will need. Our formal language framework should have two main modules: automata and grammars 3.1.

We will have a sub-module for finite automata in the automata module, where we will include representations for each finite automaton object, such as DFA (deterministic finite automaton), NFA (non-deterministic finite automaton), or ε-NFA (non-deterministic automaton with epsilon transitions). In the grammars module, we will have a representation of a CFG (context-free grammar).

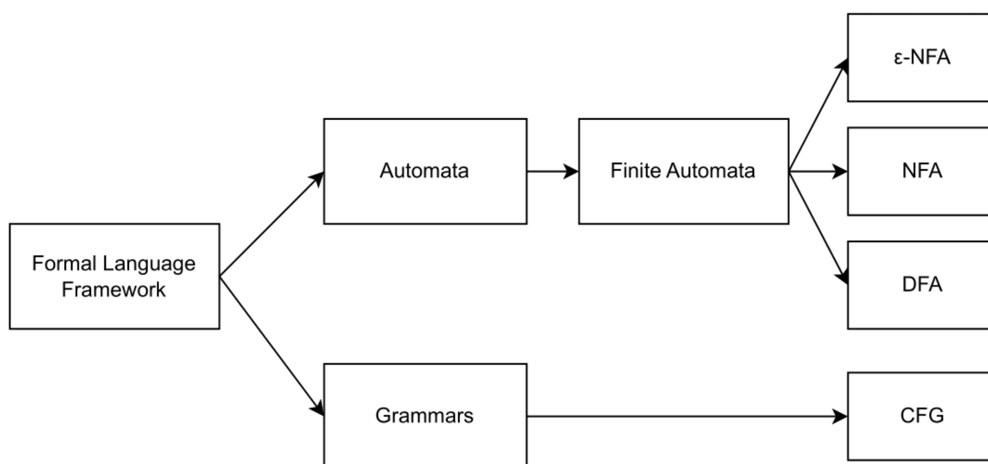


Figure 3.1: Formal Language Framework modules

Symbols and states for finite automata will be string values, and symbols for grammars will be just one-character symbols because, for testing purposes, we do not need computational power or abstraction that symbols can be anything.

3.1.1 Grammar Module

Grammar is defined by a set of non-terminal symbols, terminal symbols, a starting symbol, and a set of production rules.

We will have a class representing a symbol, and from it, we will inherit classes for non-terminal and terminal symbols. The symbol class will have one field representing the value of the symbol (most of the time, we will use just one single character value). There has to be a special class for an epsilon symbol, which will inherit from the terminal class and will have a unique constant value $. For this constant value, we have taken inspiration from the Regular Expressions Gym [20] web application.

The production rule class will consist of two main properties, which will be the body and head of the production rule. They will be a non-empty sequence of symbols.

With symbol and production rule classes established, we can define a grammar class. The grammar class will be an abstract class, and from this class, we will inherit other grammars, such as CFG, which will have additional restrictions for production rules. The grammar class will include fields for a set of non-terminals, a set of terminals, a field for the starting symbol, and a field for a set of production rules. The class diagram for the grammar module is shown in Figure 3.2.

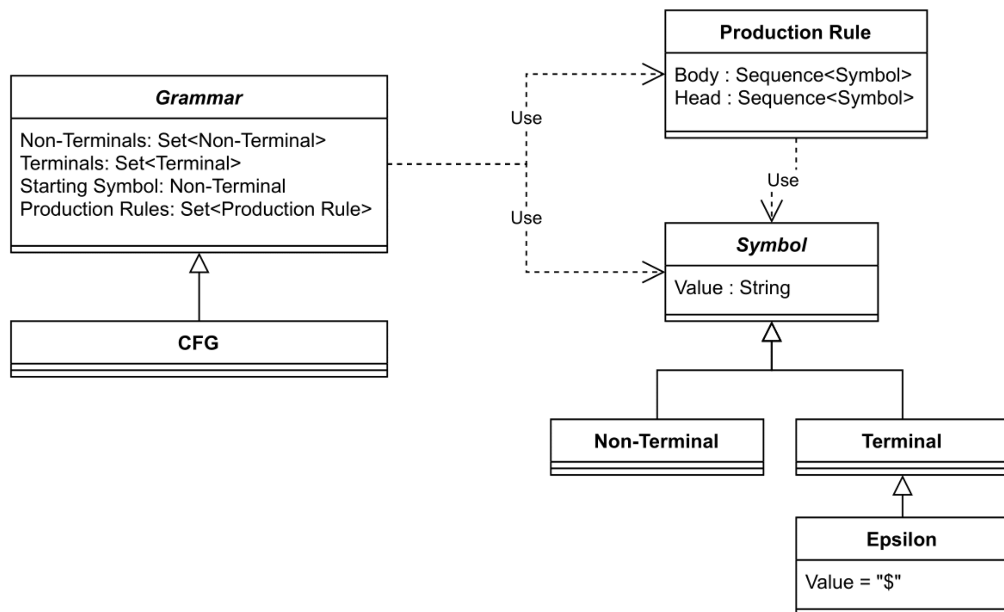


Figure 3.2: Grammars module classes

3.1.2 Finite Automata Module

The module for automata will include a class for representing all automata objects, from which a finite automaton class will be inherited.

To define a finite automaton, we need to specify an input alphabet, states, initial states, final states, and a transition function over these states and input alphabet.

Similar to the grammar module, we will have a class for a symbol with only one field representing the value of the symbol. This will not be an abstract class because automata use only one type of symbol. A special inherited class for the epsilon symbol will have its value set to $, as seen in the web application Regular Expression Gym [20].

For the state, we will create a class with one field for the string value.

The transition function will behave like a dictionary with keys as tuples of states and symbols, and values as sets of states. If the transition for a given state and symbol is not defined, it should return an empty set.

We will then define a class for a finite automaton object. This abstract class will have fields for a set of symbols (input alphabet), three sets of states (all states of the automaton, initial states, and final states), and a transition function object. From the finite automaton class, we will inherit classes for an ε-NFA, NFA, and DFA. The class diagram for the automata module is shown in Figure 3.3

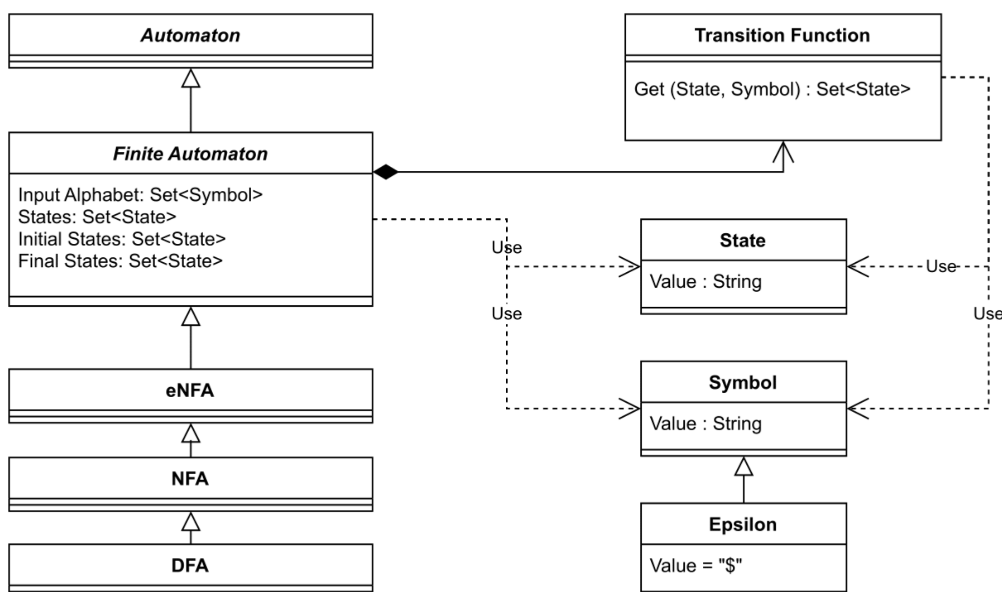


Figure 3.3: Automata module classes

3.1.3 Formal Language Object

Each formal language object, whether a grammar or a finite automaton, should be serializable and deserializable, as we will need to save them to a file and load them. Additionally, each object in this framework should have a *to-report* method, which will return the object in a human-readable form in the format corresponding to the final report, and a *to-latex* method, which will return the object as a LaTeX string.

To ensure this, we will have an abstract *formal language object* class that will serve as a predecessor for each object in our formal language framework. The complete hierarchy of classes for grammars and automata will be shown in Figure 3.4.

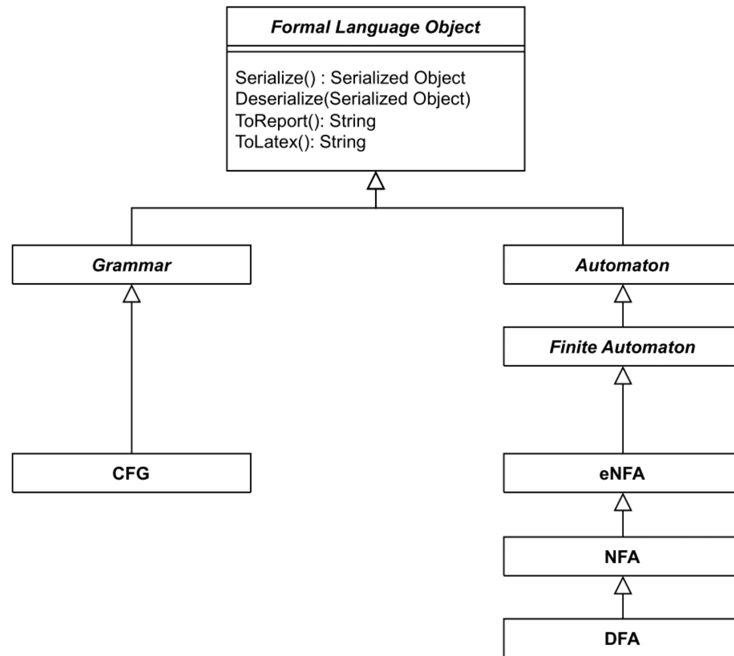


Figure 3.4: Formal Language Framework objects hierarchy

3.2 Generic Requirements

Our program will be an application for generating problem assignments from formal language theory.

- **Program Functionality**:
  - The program will generate problem assignments from formal language theory.
  - Specific properties for each problem can be set by a user. The program will then generate problem assignments with properties close to the specified ones.
  - The number of generated problem assignments can be set.
  - For each problem assignment, decorations will be created. Where decorations ensure students cannot easily copy each assignments of each other even if they have the same problem assignment. Decorations include renaming variables, symbols, states, etc.
  - The program will display the report for each generated and decorated problem assignment. The report includes the process of solving, a comparison of desired versus actual properties and the solution.
  - Our program will convert problem assignments and decorations into LaTeX format for easy inclusion into our exam for students.
- **Default Use Case**:
  - Choose a problem type to work with.

  - Set desired properties for generated problem assignments.
  - Set properties for decorations.
  - Set the number of generated problem assignments.
  - Set the number of decorations per problem assignment.
  - Generate assignments, decorate them, and create a final report.
  - Summarize the entire process into the report, including actual versus desired properties, the solving process, and solutions for each created problem assignment.
  - Convert problem assignments and decorations into LaTeX format.

- **Additional Requirements**:
  - Ability to set whether new problem assignments will be generated or it will use already generated ones. With this option, we are introducing an ability to manually change the generated problem assignments if they do not meet the required properties. This involves recalculating decorations and updating the report based on the modified assignments. Another reason to skip the generation is to change the decoration properties and only recreate decorations.
  - Ability to set whether decorations will be created.
  - Ability to set whether the final report will be created.
  - Ability to set whether problem assignments will be converted to the LaTeX format.

- **Non-Functional Requirements**:
  - Easily add new supported problem types.
  - Add new configurable properties or track new properties while solving problem assignments.

Now, let us inspect, how our program will look and how it will be structured.

3.3 Main Program

Our program will be implemented as a console application. It will read its main settings from a configuration file, supplemented with additional attributes. We will use JSON [21] for configuration files and HTML [22] with CSS [23] formatting for the final report.

3.3.1 Problem Class

Our core objective is to generate problem assignments from formal language theory, ensuring they meet required properties. Subsequently, we will create solutions for these assignments, track their actual properties, and compare them against the desired one. This process involves a generator module to create assignments

based on the specified properties, and a solver module to compute solutions while monitoring the actual properties of the generated assignments.

Initially, we will define two main components for each problem: an input structure and an output structure. Both will be subclasses of an abstract class named *problem object*. The input structure will represent the problem assignment, while the output structure will encapsulate its solution.

Each *problem object* must support serialization and deserialization since we will store and load them from files. Serialization will be handled using JSON, where the *serialize* method will return a JSON-serializable representation of the object, and the *deserialize* method will load a serialized object back into its corresponding form.

Additionally, each problem object 3.5 will provide methods for formatting: *to-html* and *to-latex*. The *to-html* method will convert the object into an HTML-formatted string suitable for inclusion in the final report, while the *to-latex* method will return the object as a LaTeX-formatted string for easy integration into test assignments. Note that problem object can be anything what is input or output of a problem. For instance, when verifying whether a string conforms to a context-free grammar using the CYK algorithm, the input consists of a context-free grammar and a string of terminals. The output in this scenario is a boolean value. This flexibility allows a problem object to be composed of nested objects or to represent a single value as required.

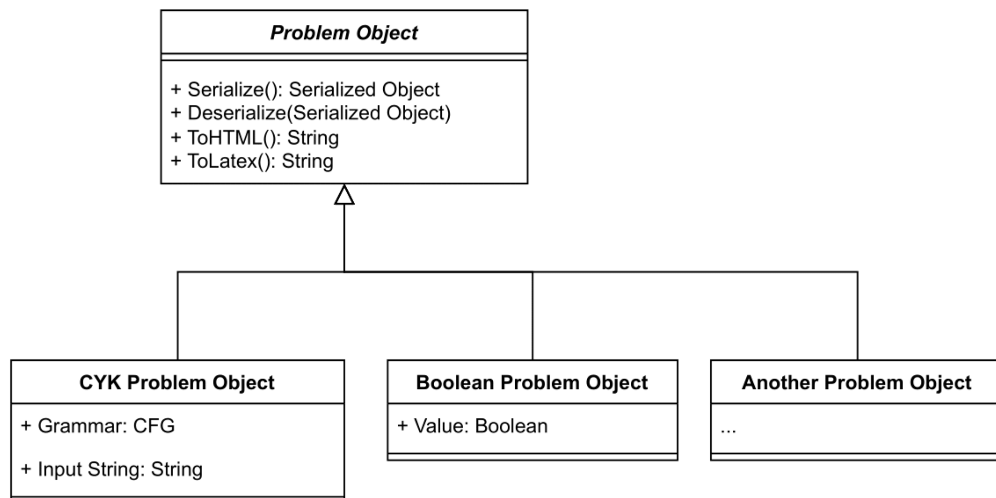


Figure 3.5: Problem Object hierarchy

Each problem will have the corresponding problem properties and configurable properties (used as input for the problem generator). Configurable properties must always be a subset of the problem properties. Each properties object will also be serializable into JSON and deserializable from JSON, allowing them to be saved into a file where a user can set them up and load them back.

Every problem must include a generator 3.6. This generator takes the configurable properties of the problem and generates an input object based on these properties. Each generator has a field containing the corresponding configurable

properties and a *generate* method that generates and returns a problem object.

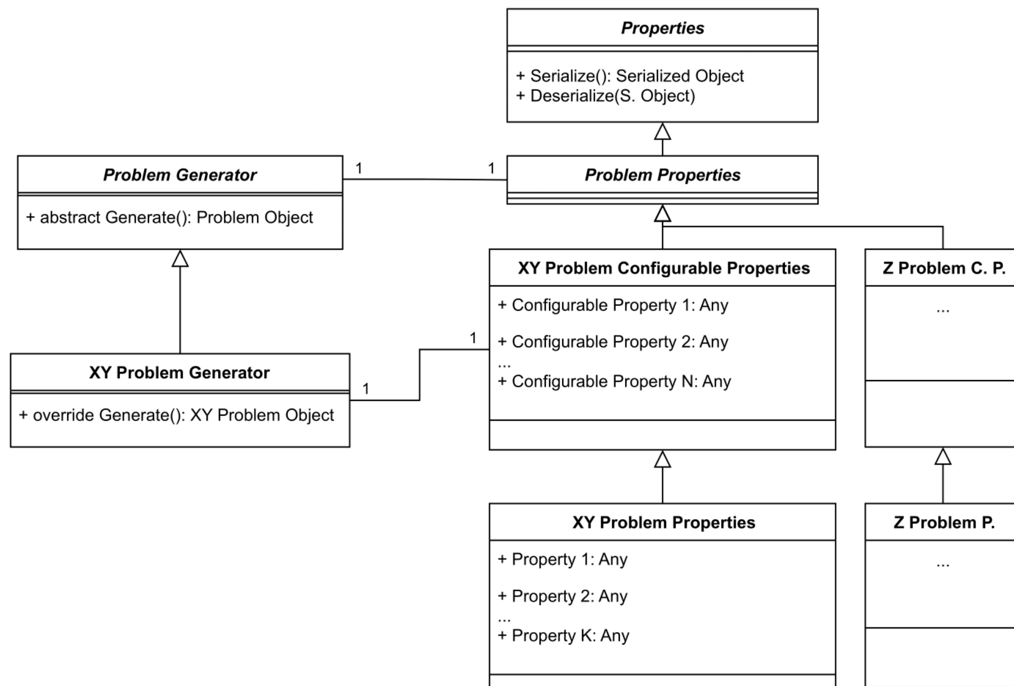


Figure 3.6: Problem Generator classes

Each type of problem object used as input for any problem must have a decorator 3.7. The decorator takes a problem object and returns the same object with renamed elements, as specified in the decorator properties. For instance, with a grammar, we can alter the names of terminals while preserving the structure of the grammar—it is akin to renaming variables in an equation.

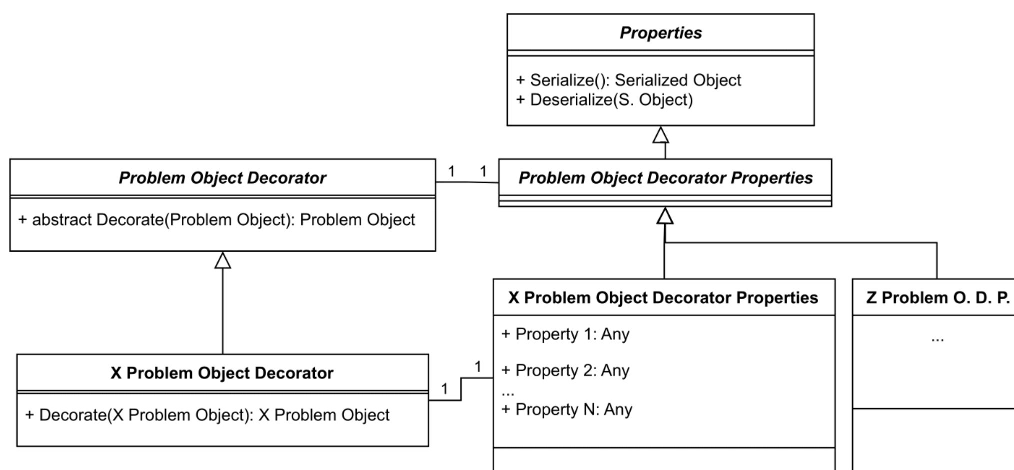


Figure 3.7: Problem Decorator classes

Finally, each problem requires a solver 3.8 that takes the input problem object,

solves it while tracking the problem properties, and returns an output problem object—that is, the solution. The solver will also create a report in the HTML format detailing the solving procedure and presenting the output object.

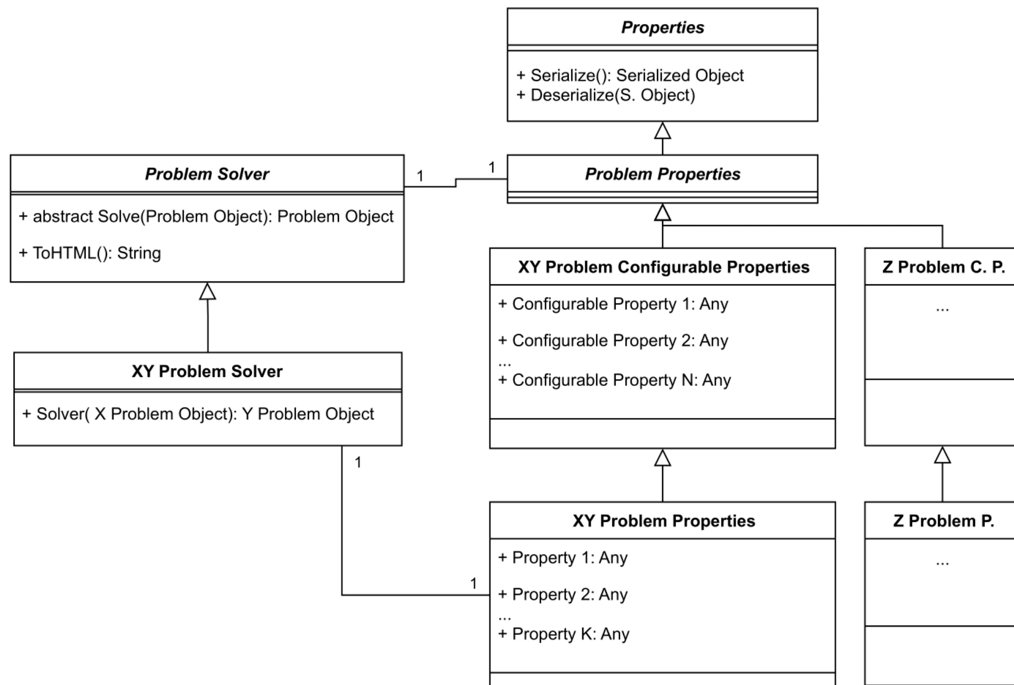


Figure 3.8: Problem Solver classes

Each problem 3.9 will be composed of a generator, decorator, and solver. These objects will be publicly accessible, along with an attribute named *input_problem_object*, where the generated and decorated problem object will be stored.

There will be three main methods for each problem: *Generate*, *Decorate*, and *GetReport*. The *Generate* method will invoke the generate method of the corresponding generator and store the generated problem object in the *input_problem_object* attribute. Subsequently, after setting the properties for the decorator, the *Decorate* method will decorate the input problem object and update the *input_problem_object* with the decorated problem object. Finally, the *GetReport* method will return an HTML string representing the problem report. This report will include the generated input object, a comparison table showing the desired configurable problem properties provided to the generator versus the actual problem properties tracked by the solver, and a detailed solving process culminating in the final solution.

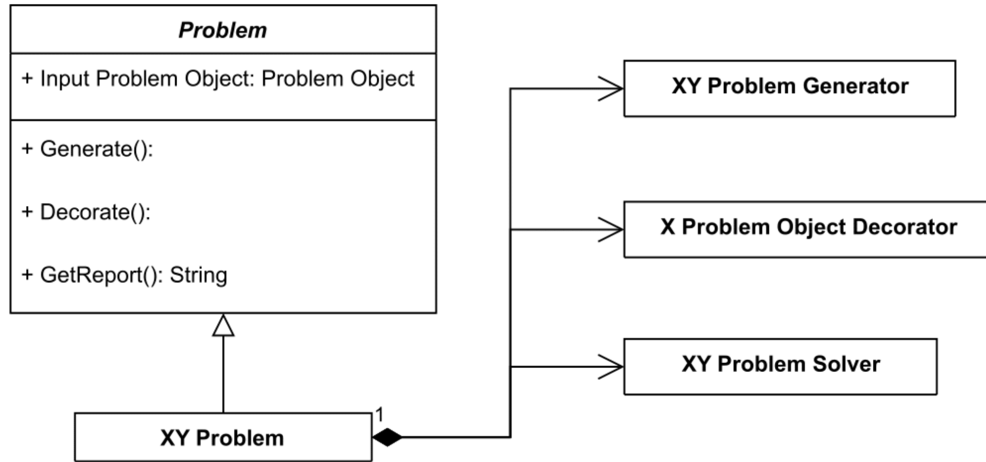


Figure 3.9: Problem classes

The *GetReport* method works as follows. Firstly, it converts the input problem object to the HTML format. Then, it utilizes the solver to solve the problem for the input problem object attribute and determines the actual problem properties. Next, it compares these actual problem properties with the desired configurable problem properties used during the generation process, obtained from the generator. This comparison is presented in an HTML table where each row represents a property, and columns display their respective values. Properties that cannot be configured but are tracked will display empty cells in the configurable properties column.

During the solving process, it retrieves the HTML string of the solving report from the solver, including the final solution. Finally, it integrates all components to compose and return the comprehensive problem report 3.10.

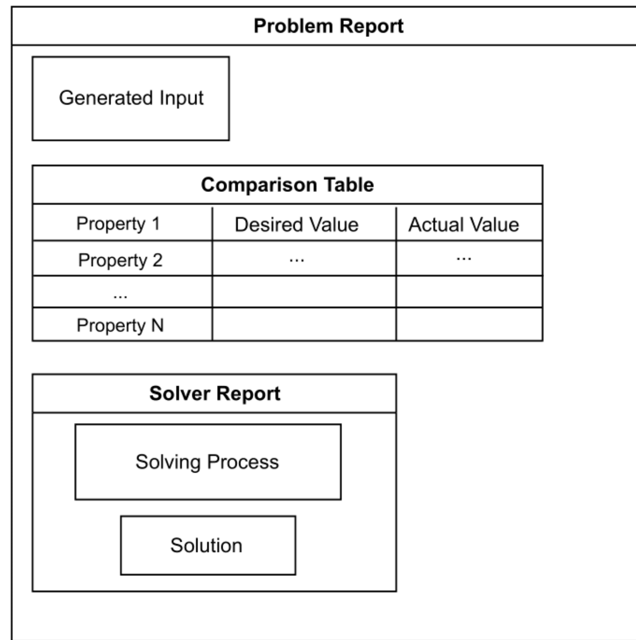


Figure 3.10: Problem report design

Our program can rely on the abstraction of a problem and its objects because the problem entity works the same for every specifically implemented problem. Each problem entity behaves uniformly, and every problem object is designed to be serializable and deserializable.

3.3.2 Console Application

Our program will be a console application. To run the program, the user will need to specify the path to a configuration file containing all settings required for a single execution.

A *job* will refer to one complete execution of our program. Each job will consist of three main parts: initialization, execution, and saving.

During the initialization, the program will set up the job with the provided properties and, if necessary, load previously generated objects and their decorations. We will discuss these use cases later.

The execution phase will primarily involve generating problem input objects, referred to as base objects, decorating them, and creating a report for each base object and its decorations. These reports will be compiled into a single comprehensive final report, which will begin with a summary of the job and then include individual reports for each object.

In the saving phase, the program will serialize the base objects and their decorations and save them into files. It will then convert them to the LaTeX format and save these files as well. Finally, the complete final report will be saved to a file.

In the beginning, we will load job properties from the file and initialize them for the job. This will include:

- Keyword for the problem type.
- Flag for saving base objects and their decorations into one file.
- Flag for saving their LaTeX versions into one file.
- Number of generated base objects (input objects for the problem).
- Number of decorations for each base object.
- Corresponding decorator properties for each decoration.
- Corresponding configurable properties for the problem generator.

A output of the job (base object files, decoration files, LaTeX files, and report file) will be stored in an output directory. By default, this directory will be named after the configuration file, but there will be an attribute for specifying a custom output directory.

There will be flags for setting up the whole process of a job. There will be flags for *generating*, *decorating*, *latex*, *report*, and *creating config* phase. They will correspond to whether the phase should be executed. Flags can be set by a user as attributes in the console when running our program.

We will start with the *creating config* phase. If this phase is to be executed, we will not load job properties from a given file. Instead, we will create default job properties for a given problem type into the specified file, allowing the user to set them up. The configuration phase can only be executed without executing other phases because the other phases require loading the configuration file with job properties.

If the *generating* phase is to be executed, the base objects will be generated by a corresponding problem generator during the execution part of the job. If it is not executed, we assume that a job with the same properties has already taken place, and we will load the already generated base objects during the initialization part.

Similarly, for the *decorating* phase, if it is to be executed, we will decorate the created or loaded base objects during the execution part of the job. If it is not executed, we assume that they were already created in a previous job, and we will load them from their files during the initialization part.

If the *latex* phase is to be executed, we will convert all base objects and their decorations to LaTeX and save them to files, considering the property of whether they should be stored in one file or not.

Lastly, in the *report* phase, we will create a final report and store it in a file if this phase is to be executed, and skip it if it is not. The final report file name can also be customized in the attributes.

The phases to be executed should be specified by running our program with the appropriate attributes. Each time, it must be run with the required attribute for the job file name. A job file contains the job properties, and the name of this file will also serve as the name of the job. Additionally, there will be optional attributes for phases. If no optional attributes are provided, the program should run with default behavior, which includes the *generating*, *decorating*, *latex*, and *report* phases. If the attribute for the *create config* phase is provided, a problem

type keyword must also be provided so the program knows for which problem to create job properties. Two other optional attributes are available: one for setting the name of the output directory and another for setting the name of the final report file.

Our program should print the progress of the job to the console, indicating what happened, which processes were successful, which failed, and if any errors occurred.

Finally, each property of a problem, problem object decorator, or job must be serializable to JSON and deserializable. This allows conversion to a file where users can set them up. Each decorator or problem object decorator will need unique corresponding properties for creating the decoration. The same applies to each problem and its problem properties.

3.3.3 Non-Functional Requirements

The main non-functional requirements are that we can easily add new problems for our program to support. If we want to add a new problem, we will need to create custom problem objects for the input and output of the problem (or use the existing ones), implement a generator and define configurable properties for the problem, implement a solver and define all tracked properties for the problem, and finally, create a decorator for the input problem object with defined properties for the decorator. Once these components are created, they should be integrated into the new problem. The final step of adding the newly defined problem to the list of supported problems will be straightforward.

The functionality of the program should not depend on the specific type of problem. This means that using our program will be consistent across all problem types.

4. Problems

In this chapter, we analyze selected problems that our generator will produce, detailing which properties can be configured for each problem and other properties that we are tracking. We will also analyze how to generate a problem with the desired properties. The problems included are:

- Determinization of NFA Problem: Converting a NFA (non-deterministic finite automaton) to an equivalent DFA (deterministic finite automaton).
- CYK Problem: Determining if a string is accepted by a CFG (context-free grammar) using the CYK (Cocke–Younger–Kasami) algorithm.
- Proper CFG Problem: Converting a CFG (context-free grammar) to a proper CFG.

4.1 Determinization of NFA Problem

As mentioned, the input for this problem will be an NFA (without epsilon transitions) which we will convert to an equivalent DFA. It is known that for every NFA, there exists an equivalent DFA, and the two automata are considered equivalent if they accept the same language.

4.1.1 Algorithm

The algorithm for a determinization is shown in Figure 4.1:

Require: NFA $M = (Q, \Sigma, \delta, I, F)$ where $I \subseteq Q$ (multiple initial states)
Ensure: DFA $M' = (Q', \Sigma, \delta', q'_0, F')$ such that $L(M) = L(M')$

1: $q'_0 \leftarrow I$
2: $Q' \leftarrow \{q'_0\}$
3: **for all** $q' \in Q'$ **do**
4:     **for all** $a \in \Sigma$ **do**
5:         $\delta'(q', a) \leftarrow \bigcup_{p \in q'} \delta(p, a)$
6:         $Q' \leftarrow Q' \cup \delta'(q', a)$
7:     **end for**
8: **end for**
9: $F' \leftarrow \{q' \in Q' \mid q' \cap F \neq \emptyset\}$
10: $M' \leftarrow (Q', \Sigma, \delta', q'_0, F')$
11: **return** M'

Figure 4.1: Algorithm for determinization of NFA

This algorithm supports NFAs with initial states. Note that the number of the initial states in the input NFA will be one of the configurable properties of this problem.

The algorithm begins by initializing the initial state of the DFA, q'_0, to be the set of all initial states of the NFA, I. This is because the DFA must start in a

state that represents all possible initial states of the NFA. The set of DFA states, Q', is then initialized to contain only this initial state q'_0.

Next, the algorithm iteratively processes each new state q' in Q'. For each state q', it examines each possible input symbol a. The transition function δ' of the DFA is then defined by computing the set of states that the NFA can transition to from any state in q' on input a. This computed set is represented as $\delta'(q', a)$. This new set of states, $\delta'(q', a)$, representing a new state, which is added to the set of DFA states Q'.

After processing all states and symbols from the input alphabet, the algorithm determines the set of accepting states F' of the DFA. A state q' in the DFA is considered an accepting state if there is at least one state in q' that is an accepting state in the NFA. Formally, F' is defined as the set of all states q' in Q' for which the intersection with F (the set of accepting states of the NFA) is non-empty.

Finally, the resulting DFA M' is defined by tuple $(Q', \Sigma, \delta', q'_0, F')$, encapsulating the newly constructed states, transition function, initial state, and accepting states. This DFA M' is guaranteed to accept the same language as the original NFA M, effectively completing the conversion process.

4.1.2 Properties

Now, let us outline all the properties we want to track or require for this problem. We will begin with some fundamental properties:

- Number of symbols in the input alphabet: $|\Sigma|$
- Number of states: $|Q|$
- Number of initial states: $|I|$
- Number of final states: $|F|$

These properties pertain to the input NFA. Additionally, there are three properties that may be challenging to configure during problem generation but are worth to track:

- Number of new states in the resulting DFA: $|Q'|$
- Two new states with initial states from the input NFA occurred
- Occurrence of an empty set state, or ”garbage” state, in the DFA

The number of new states in the DFA will equal the number of combinations of states in the new DFA. Tracking this property is crucial because the state count in the resulting DFA can grow exponentially because number of these combinations can be up to $2^{|Q|}$. This property helps us gauge the computational demand of solving the problem and assess whether the generated assignment is practical for student use. We aim to ensure that students do not spend excessive time on this problem. Their ability to solve it can be demonstrated even with smaller automata, where the creation of numerous new states is unnecessary.

The property of *two new states with initial states* refers to a scenario in which the DFA may contain two new states that each include initial states from the

NFA. Students may incorrectly identify both as new initial states for the DFA, which violates determinism since a DFA can only have one initial state.

The *garbage* state or empty set state in the DFA is a new state created to handle undefined transitions in the NFA. For instance, in the following NFA:

		0	1
$\rightarrow$	A		B
$\leftarrow$	B	A	

There is no defined transition for state A and symbol 0. During determinization, a new empty set state is created to handle such cases, and it is added to the DFA. Consequently, every transition that is not defined in the NFA leads to this empty set state in the output DFA. The resulting DFA would look like this:

		0	1
$\rightarrow$	$\{A\}$	$\{\}$	$\{B\}$
$\leftarrow$	$\{B\}$	$\{A\}$	$\{\}$
	$\{\}$	$\{\}$	$\{\}$

Tracking these properties ensures that the problem assignments we generate are diverse and sufficiently challenging, yet not overly complex. Some of these properties will be integral for generating the input NFA.

4.1.3 Generation Process

Let us analyze the process of generating an input NFA, focusing primarily on the creation of the transition function.

To begin, we establish configurable properties that will influence the generation process. It is important to note that these configurable properties represent only a subset of all properties tracked for the problem. For the determinization process, the configurable properties include:

- Number of symbols in the input alphabet: $|\Sigma|$
- Number of states: $|Q|$
- Number of initial states: $|I|$
- Number of final states: $|F|$

These properties define the characteristics of the generated input NFA.

The generation of an input NFA $M = (Q, \Sigma, \delta, I, F)$ starts with creating Q and Σ. The naming of symbols and states does not matter initially, as they can be changed later. They just need to be unique. The size of Σ is determined by the *number of symbols in the input alphabet* property, and the size of Q is determined by the *number of states* property.

Next, we define the initial states I and final states F. These will be random subsets of states Q, with the size of I given by the *number of initial states* property and the size of F given by the *number of final states* property.

After creating the states, input alphabet, initial states, and final states, we define the transition function δ. This generation is based on a random strategy with certain constants to produce a reasonable transition function.

The transition function δ is generated by iterating over all states and symbols. For each state-symbol pair, we determine whether the transition should be empty using a biased random choice. If the transition is not empty, the number of new states for the transition is chosen randomly with an exponentially decreasing probability. This ensures that transitions with fewer states are more likely.

Specifically, the probability $P(k)$ of choosing $k \in \{1, 2, 3, \ldots, |Q|\}$ states for the transition follows an exponential distribution. To ensure the probabilities sum to 1 over the possible values of k, we use the following normalization:

$$P(k) = \frac{e^{-\lambda k}}{\sum_{j=1}^{|Q|} e^{-j}}$$

The parameter λ in the exponential distribution controls the rate of decay of probabilities. A higher value of λ results in a faster decrease in the likelihood of choosing larger values, effectively emphasizing transitions with fewer states.

For $k \in \{1, 2, 3, 4, 5\}$ and $\lambda = 0.6$, the probabilities are:

$$\begin{aligned}
P(1) &= e^{-0.6 \cdot 1} &\approx 0.5488 \\
P(2) &= e^{-0.6 \cdot 2} &\approx 0.3012 \\
P(3) &= e^{-0.6 \cdot 3} &\approx 0.1653 \\
P(4) &= e^{-0.6 \cdot 4} &\approx 0.0907 \\
P(5) &= e^{-0.6 \cdot 5} &\approx 0.0498
\end{aligned}$$

These probabilities ensure that the likelihood of choosing more states decreases exponentially, which means that we can set the decaying factor somewhere between 0.6 and 1.0.

4.2 CYK Problem

The next problem, now from grammars, will be determining if a string can be generated by a CFG. For this, we will use the CYK algorithm, which takes as input a CFG in Chomsky normal form and the input string, and returns `true` or `false` depending on whether the input string is in the language generated by the grammar or not.

4.2.1 Algorithm

The CYK algorithm is shown in Figure 4.2

Require: CFG $G = (N, \Sigma, P, S)$ in Chomsky normal form, $x = x_1 x_2 \ldots x_n \in \Sigma^*$, $x_i \in \Sigma$, $n \in \mathbb{N}$
Ensure: Boolean value `true` if $x \in L(G)$, else `false`.

```
 1: P[i, j] ← ∅, ∀i, j ∈ {1, 2, ..., n}                          ▷ Initialize array
 2: for i ∈ {1, ..., n} do                ▷ Initialize for substrings of length 1
 3:     P[n − (i − 1), i] ← P[n − (i − 1), i] ∪ {A}, ∀A ∈ N, (A → x_i) ∈ P
 4: end for
 5: for j ∈ {2, ..., n} do                                  ▷ Length of substring
 6:     for i ∈ {1, ..., n − j + 1} do                       ▷ Start of substring
 7:         for k ∈ {1, ..., j − 1} do                   ▷ Partition of substring
 8:             a ← (n − (i − 1) − (j − 1))                   ▷ Current row index
 9:             b ← i                                      ▷ Current column index
10:             P[a, b] ← P[a, b] ∪ {A}, ∀{A → BC} ∈ P, B ∈ P[a + k, b] ∧ C ∈
    P[a, b + (j − 1) − (k − 1)]
11:         end for
12:     end for
13: end for
14: if S ∈ P[1, 1] then
15:     return true                                                  ▷ x ∈ L(G)
16: else
17:     return false                                                 ▷ x ∉ L(G)
18: end if
```

Figure 4.2: CYK Algorithm

The CYK algorithm operates by constructing a table where each entry $P[i, j]$ represents the set of non-terminal symbols that can generate the substring of length $n - (j - 1) - (i - 1)$ starting at position j in the input string. This table is initialized and then filled iteratively with using the production rules of the CFG.

Initially, the table is filled for substrings of length 1. For each position i in the input string, if there is a production rule in the grammar where a non-terminal A produces the terminal symbol at position i, then A is added to cell $P[n - (i - 1), i]$ on the main diagonal of the table. This step ensures that the table correctly represents all possible single-symbol substrings according to the grammar.

The algorithm then proceeds to fill the table for substrings of increasing lengths, from 2 up to n, where n is the length of the input string. For each substring of length j, and for each possible starting position i, the algorithm considers all possible partitions of the substring into two smaller substrings. For each partition, it checks whether there are production rules in the grammar that can generate the entire substring from the non-terminal symbols that generate the smaller substrings. If such production rules exist, the corresponding non-terminal symbol is added to the corresponding cell $P[n - (i - 1) - (j - 1), i]$.

Specifically, for each substring α of length j starting at position i, the algorithm considers each partition of this substring into two parts of lengths $j - k$ and k. It then checks if there exists a rule $A \to BC$, where B are all possible non-terminals generating the first part of the substring α of length $j - k$ and C are all possible non-terminals generating the second part of the substring α of length k. If such a rule exists, A is added to the cell representing all possible

non-terminals generating the substring α. This process is repeated for all possible splits of the substring α, ensuring that the table correctly represents all possible ways to generate substrings of increasing lengths.

Finally, after filling the table, the algorithm checks whether the start symbol S of the grammar is in $P[1,1]$, which represents all possible non-terminals that can generate the whole input string. If S is present in this cell, the input string is generated by the grammar; otherwise, it is not.

Example (CYK Algorithm). Let us demonstrate this algorithm on an example. For input string `baaba` and grammar $G_1 = (\{S, A, B, C\}, \{\texttt{a}, \texttt{b}\}, P, S)$,

$$
\begin{aligned}
P = \{ & \\
& S \to AB \mid BC, \\
& A \to BA \mid \texttt{a}, \\
& B \to CC \mid \texttt{b}, \\
& C \to AB \mid \texttt{a} \\
\} &
\end{aligned}
$$

First, we initialize the table and fill it with empty sets. Then, we fill the main diagonal of the table for substrings of length 1, as shown below. Note that "-" in a cell means that the algorithm does not work with that cell. If the cell is empty, it contains an empty set $\{\}$.

	1	2	3	4	5
1					$\{A, C\}$
2				$\{B\}$	-
3			$\{A, C\}$	-	-
4		$\{A, C\}$	-	-	-
5	$\{B\}$	-	-	-	-

As we can see, for example, cell $P[4,2] = \{A, C\}$ corresponds to the substring `a` in the input string `baaba`. The `a` can be generated by production rules $A \to \texttt{a}$ and $C \to \texttt{a}$, which corresponds to the set $\{A, C\}$ in the cell.

Next, we fill one diagonal after another. After filling the main diagonal, we fill the next diagonal for substrings of length 2.

	1	2	3	4	5
1				$\{A, S\}$	$\{A, C\}$
2			$\{S, C\}$	$\{B\}$	-
3		$\{B\}$	$\{A, C\}$	-	-
4	$\{A, S\}$	$\{A, C\}$	-	-	-
5	$\{B\}$	-	-	-	-

We continue this process until we get to the cell $P[1,1]$.

	1	2	3	4	5
1	$\{A, C, S\}$	$\{A, C, S\}$	$\{B\}$	$\{A, S\}$	$\{A, C\}$
2		$\{B\}$	$\{S, C\}$	$\{B\}$	-
3		$\{B\}$	$\{A, C\}$	-	-
4	$\{A, S\}$	$\{A, C\}$	-	-	-
5	$\{B\}$	-	-	-	-

For demonstration, let us show how to fill cell $P[1,2] = \{A, S\}$, which corresponds to the substring starting at the second position of length 4 (`baaba`), meaning $\alpha =$ `aaba`. For each split $k \in \{1, 2, 3\}$, we check the corresponding cells and combinations of non-terminals in them.

- For $k = 1$, we split `aaba` into $\alpha_1 =$ `aab` and $\alpha_2 =$ `a`. For α_1, we have the corresponding cell $P[2,2] = \{B\}$ and for α_2, we have the cell $P[1,5] = \{A, C\}$. All possible pairs are BA, BC. For these pairs, we have production rules $A \to BA$ and $S \to BC$. Thus, we add A and S to cell $P[1,2]$.
- For $k = 2$, we get $\alpha_1 =$ `aa` and $\alpha_2 =$ `ba`. The corresponding cells are $P[3,2] = \{B\}$ and $P[1,4] = \{A, S\}$. All possible pairs are BA, BS. Only A generates BA, so we add it to $P[1,2]$.
- For $k = 3$, $\alpha_1 =$ `a` and $\alpha_2 =$ `aba`. We inspect $P[4,2] = \{A, C\}$ and $P[1,3] = \{B\}$. All possible pairs are AB, CB, so we add S and C to the cell.

The cell $P[1,2]$ will be equal to $\{A, C, S\}$.

□

With understanding of the algorithm, we can inspect the tracked properties.

4.2.2 Properties

For this problem, we will consider more straightforward properties that will also be used during the generation process. These properties are:

- Lenght of the input string
- Number of terminals
- Number of non-terminals
- If the starting symbol is generating epsilon

All of these properties can be easily tracked and fulfilled during the generation process.

Other properties that are not as straightforward or are harder to fulfill during the generation process include:

- Whether the input string is accepted
- Count of empty cells
- Work coefficient

The property of *whether the input string is accepted* is easy to track, but fulfilling it during the generation would involve solving the problem. If the property is not fulfilled, the generation process would have to be rerun, which could result in an endless loop due to the random nature of the generation. An alternative way to fulfill this property would be to work backwards when filling the CYK table. To ensure the input string is accepted, we would place the starting symbol

in the cell representing the whole string and then randomly fill the table in reverse order. This approach, however, could lead to countless possibilities for filling the table, many of which would result in unusable problem assignments, making this option impractical.

The next property is the *count of empty cells*. This is the number of empty cells in the CYK table, considering only cells that are used, which are those above and including the main diagonal. Controlling this during the generation would again involve solving the problem and then regenerating it, which, as mentioned, could lead to an endless process.

Finally, we have the *work coefficient*, which is a non-negative whole number corresponding to how demanding it is to solve the generated assignment. We will count this as the number of comparisons. When filling the table for substrings of length 1, we add 1 for each comparison of a production rule with the terminal (substring of length 1) if the production rule generates it. Similarly, in the second half of the algorithm, we add 1 for each comparison of a pair of non-terminals versus a production rule. This property can show that two almost similar problem can be very differently hard to solve, making it unfair to assign them to different groups of students.

These properties will give us good insight into the solving process and allow us to determine which problem assignments are suitable.

4.2.3 Generation Process

The generation of problem assignments for the CYK problem will be based on randomization, where we will initially fulfill configurable properties and then create production rules based on biased randomization.

First, for a CFG $G = (N, \Sigma, P, S)$, we need to define terminals Σ, non-terminals N, and the starting symbol S of our desired CFG in CNF form. The numbers of terminals and non-terminals are specified in the properties. Then, we will randomly choose the starting symbol, or we can simply pick the first one. In most cases, we will name it S at the end. Again, the naming does not matter. Then we create production rules.

Firstly, we must ensure that each terminal can be generated by the CFG. However, we do not want to have many non-terminals that generate the same terminal. For terminal t, we will set the number k_t of non-terminals that can generate it. The number will be $k_t \in \{1, 2, \dots, m\}$. This means we do not want more than m non-terminals generating the terminal. The m will be a constant set to a small natural number, for example, $m = 3$. We will then choose the k_t randomly with an exponentially decreasing chance for higher values. This means we will most likely get only 1 non-terminal that can generate terminal t. After setting the number of non-terminals that can generate t, we will randomly choose a subset of non-terminals $K_t \subseteq N$, where $|K_t| = k_t$ (if there are not enough non-terminals, we will allow $|K_t| \leq k_t$), and create production rules for each non-terminal in K_t, $P \leftarrow P \cup \{A \to t \mid A \in K_t\}$. We repeat this process for each terminal $t \in \Sigma$.

After ensuring that each terminal can be generated, we will add the rule $S \to \epsilon$ if it is required in the properties.

We can finally add rules where the right-hand side is composed from a pair of

non-terminals. Again, we do not want many rules for each non-terminal because then it will easily happen that each cell of the table will include all non-terminals. For non-terminal A, we will set the number k_A, which will again be between 1 and some constant m, which we will set to $m = 3$. The k_A will then be randomly chosen from $\{1, 2, \ldots, m\}$ with a decreasing exponential probability, which will not have a large descending factor λ because we do not want only one rule for each non-terminal. Consequently, we will create a new set of rules $P_A = \{A \to BC \mid B, C \in N\}$, where $|P_A| = k_A$ and B, C are randomly chosen non-terminals. In other words, we will create k_A new random rules in the format of $A \leftarrow BC$. Then we will add them to all production rules, $P \leftarrow P \cup P_A$. This process will be repeated for each non-terminal $A \in N$.

This generation process will not always produce reasonable assignments, but we can iterate it many times and easily determine if the generated problem assignment is reasonable by looking at the CYK table and the work coefficient.

4.3 Proper CFG Problem

The third and most complex problem is converting a CFG to a proper CFG. Recall that a CFG is proper if it is cycle-free 11, ε-rule-free, and without redundant symbols 8. The input of the problem is a CFG, and the output is an equivalent CFG that is proper. Transforming a CFG into a proper CFG involves several subproblems: excluding redundant symbols, ε-rules, and simple rules. We will introduce algorithms for these subproblems.

4.3.1 Algorithm for Exclusion of Redundant Symbols

To get a proper CFG, we need to remove redundant symbols. A redundant symbol is any symbol that is either non-generating 6 or unreachable 7. Therefore, we need to remove both non-generating and unreachable symbols. Both algorithms take a CFG as input and output a CFG.

We start with the algorithm for excluding non-generating symbols 4.3.

Require: CFG $G = (N, \Sigma, P, S)$
Ensure: CFG $G' = (N', \Sigma, P', S)$ such that $L(G') = L(G)$, where N' includes only generating symbols

1: $N_0 \leftarrow \emptyset$; $i \leftarrow 0$
2: **repeat** ▷ Finding generating symbols
3: $\quad i \leftarrow i + 1$
4: $\quad N_i \leftarrow \{A \mid A \in N, (A \to \alpha) \in P, \alpha \in (N_{i-1} \cup \Sigma)^*\} \cup N_{i-1}$
5: **until** $N_i = N_{i-1}$
6: $N' \leftarrow N_i$
7: $P' \leftarrow \{A \to \alpha \mid A \in N', \alpha \in (N' \cup \Sigma)^*, (A \to \alpha) \in P\}$ ▷ Removing rules with non-generating symbols
8: $G' \leftarrow (N', \Sigma, P', S)$ **return** G'

Figure 4.3: Algorithm for exclusion of non-generating symbols from CFG

The algorithm, starts by initializing an empty set N_0 and iteratively computes sets N_i until convergence, where N_i consists of all non-terminal symbols that can

generate strings composed solely of terminal symbols and previously identified generating non-terminals. Subsequently, the resulting set N' represents the set of all generating non-terminals. The algorithm then constructs a new CFG G' by retaining only those production rules from G where both the left-hand side non-terminal and the right-hand side production consist solely of symbols from N' and Σ.

The algorithm for exclusion of unreachable symbols is in Figure 4.4.

Require: Context-free grammar $G = (N, \Sigma, P, S)$
Ensure: CFG $G' = (N', \Sigma', P', S)$ such that $L(G') = L(G)$, where N' and Σ' includes only reachable symbols
1: $V_0 \leftarrow \{S\}$; $i \leftarrow 0$
2: **repeat** ▷ Finding reachable symbols
3: $\quad i \leftarrow i + 1$
4: $\quad V_i \leftarrow \{X \mid X \in N \cup \Sigma, (A \to \alpha X \beta) \in P, A \in V_{i-1}, \alpha, \beta \in (N \cup \Sigma)^*\} \cup V_{i-1}$
5: **until** $V_i = V_{i-1}$
6: $N' \leftarrow V_i \cap N$
7: $\Sigma' \leftarrow V_i \cap \Sigma$
8: $P' \leftarrow \{A \to \alpha \mid A \in N', \alpha \in V_i^*, (A \to \alpha) \in P\}$ ▷ Removing rules with unreachable symbols
9: $G' \leftarrow (N', \Sigma', P', S)$
10: **return** G'

Figure 4.4: Algorithm for exclusion of unreachable symbols from CFG

The algorithm operates on the principle of breadth-first search (BFS), commencing from the start symbol. Initially, the algorithm initializes V_0 with the start symbol S and iteratively expands V_i, where V_i includes symbols reached by i steps of the algorithm. This process continues until V_i stabilizes, ensuring all reachable symbols are identified. Subsequently, N' and Σ' are determined as the intersections of V_i with N and Σ, respectively. The algorithm then constructs P' by retaining only those productions from P where are no unreachable symbols.

The final algorithm 4.5 for the exclusion of redundant symbols uses these two algorithms.

Require: Context-free grammar $G = (N, \Sigma, P, S)$
Ensure: CFG $G'' = (N'', \Sigma'', P'', S)$ such that $L(G') = L(G)$, without redundant symbols
1: Using algorithm *exclusion of non-generating symbols* over G, we remove all non-generating symbols and we get an output CFG G'.
2: Using algorithm *exclusion of unreachable symbols* over G', we remove all unreachable symbols and we get $G'' = (N'', \Sigma'', P'', S)$.
3: **return** G''

Figure 4.5: Algorithm for exclusion of redundant symbols from CFG

The order of these two algorithms is crucial. The exclusion of non-generating symbols may create new unreachable symbols, but not vice versa. If done in the reverse order, some new unreachable symbols might not be excluded, thus failing to eliminate all redundant symbols and the algorithm would be incorrect.

4.3.2 Algorithm for Exclusion of ε-rules

The next algorithm required to make a CFG proper is exclusion of ε-rules 4.6.

Require: CFG $G = (N, \Sigma, P, S)$
Ensure: CFG $G' = (N', \Sigma, P', S')$ without ε-rules, such that $L(G') = L(G)$
1: $N_0 \leftarrow \emptyset$; $i \leftarrow 0$
2: **repeat** ▷ BFS for nonterminals X, $X \Rightarrow^* \varepsilon$
3: $\quad i \leftarrow i + 1$
4: $\quad N_i \leftarrow \{A \mid (A \rightarrow \alpha) \in P, \alpha \in N_{i-1}^*\}$
5: **until** $N_{i-1} = N_i$
6: $N_\varepsilon \leftarrow N_i$
7: $P' \leftarrow \{A \rightarrow \alpha_1\alpha_2 \dots \alpha_n \mid (A \rightarrow X_1X_2 \dots X_n) \in P, X_i \in N \cup \Sigma,$
8: $\quad (\alpha_i = X_i \text{ or } \varepsilon \text{ if } X_i \in N_\varepsilon),$
9: $\quad (\alpha_i = X_i \text{ if } X_i \notin N_\varepsilon),$
10: $\quad \alpha_1\alpha_2 \dots \alpha_n \neq \varepsilon\}$ ▷ No ε-rules
11: **if** $S \in N_\varepsilon \wedge \exists(A \rightarrow \alpha S \beta) \in P' \mid \alpha, \beta \in (N \cup \Sigma)^*, A \in N$ **then**
12: $\quad P' \leftarrow P' \cup \{S' \rightarrow \varepsilon, S' \rightarrow S\}$ ▷ Introduce S' not in N
13: $\quad N' \leftarrow N \cup \{S'\}$
14: **else if** $S \in N_\varepsilon$ **then**
15: $\quad S' \leftarrow S$
16: $\quad P' \leftarrow P' \cup \{S' \rightarrow \varepsilon\}$
17: **else**
18: $\quad S' \leftarrow S$
19: **end if**
20: $G' \leftarrow (N', \Sigma, P', S')$
21: **return** G'

Figure 4.6: Algorithm for exclusion of ε-rules from CFG

This algorithm transforms a given CFG $G = (N, \Sigma, P, S)$ into a new CFG $G' = (N', \Sigma, P', S')$ that has no ε-rules (except optionally for the starting symbol), ensuring that $L(G') = L(G)$. The algorithm begins by initializing N_0 as an empty set and iteratively expands N_i using a breadth-first search (BFS) approach to find nonterminals X such that $X \Rightarrow^* \varepsilon$. This process continues until N_i stabilizes, indicating that all nonterminals that can derive ε symbol have been identified. Next, the set N_ε is set to N_i. The set of productions P' is then constructed by modifying the original productions in P to exclude ε-rules, ensuring no ε-productions remain. If the start symbol S can derive ε and appears on the right side of any rule, a new start symbol S' is introduced with the production rules $S' \rightarrow \varepsilon$ and $S' \rightarrow S$, and S' is added to the set of nonterminals N. Otherwise, S' is set to S. The resulting CFG $G' = (N', \Sigma, P', S')$ is returned, which is equivalent to the original CFG but without ε-rules.

4.3.3 Exclusion of Simple Rules Algorithm

Finally, we have the algorithm for exclusion of simple rules from a CFG 4.7.

Require: CFG $G = (N, \Sigma, P, S)$
Ensure: CFG $G' = (N, \Sigma, P', S')$ without simple rules, such that $L(G') = L(G)$

```
 1: for A ∈ N do
 2:     N_0 ← {A}; i ← 0
 3:     repeat
 4:         i ← i + 1
 5:         N_i ← {C | (B → C) ∈ P, B ∈ N_{i−1}} ∪ N_{i−1}
 6:     until N_{i−1} = N_i
 7:     N_A ← N_i
 8: end for
 9: P' ← ∅
10: for A ∈ N do
11:     P' ← P' ∪ {A → α | (B → α) ∈ P, B ∈ N_A, α ∈ ((N ∪ Σ)* \ N)}
12: end for
13: G' ← (N, Σ, P', S')
14: return G'
```

Figure 4.7: Algorithm for exclusion of simple rules from CFG

Initially, for each nonterminal A in N, the algorithm constructs the set N_A which contains all nonterminals reachable from A through a sequence of simple rules (productions of the form $B \to C$ where $B, C \in N$). This is done using an iterative process similar to breadth-first search (BFS), starting from $N_0 = A$ and expanding it until no new nonterminals are added using only simple rules. Once the sets N_A are computed for all nonterminals A, the algorithm constructs the new set of productions P' by including a production $A \to \alpha$ for each original production $B \to \alpha$ where $B \in N_A$ and α is a string of terminals and nonterminals that does not contain any simple rules. Finally, the algorithm returns the new grammar $G' = (N, \Sigma, P', S)$, which is equivalent to the original grammar but without simple rules.

4.3.4 Proper CFG Algorithm

We now focus on an algorithm that will convert a CFG into a proper CFG. According to Theorem1, a CFG is cycle-free if there are no ε-rules and simple rules. Therefore, to convert the CFG to a proper CFG, we must:

1. Exclude ε-rules,
2. Exclude simple rules,
3. Exclude redundant symbols.

This order is crucial because exclusion of ε-rules can create new simple rules, and both processes of exclusion of ε-rules and simple rules can introduce new redundant symbols. Therefore, redundant symbols must be excluded at the end of the process. This ensures that the new CFG is cycle-free, ε-rules-free, and devoid of redundant symbols, resulting in a proper CFG.

However, to save time and effort when performing this process manually, we will modify the order slightly:

1. Exclude redundant symbols,
2. Exclude ε-rules,
3. Exclude simple rules,
4. and exclude redundant symbols again.

While exclusion of ε-rules and simple rules can introduce new redundant symbols, they cannot make a redundant symbol non-redundant. This modified process still results in a proper CFG but is more efficient when doing it manually. Exclusion of redundant symbols initially can prevent unnecessary work by avoiding the creation of new rules for symbols that will ultimately be removed. Thus, this composed process of invoking the defined algorithms will yield a proper CFG with various properties that can be tracked or configured.

4.3.5 Properties

This problem involves the largest number of properties, which will be divided into parts, each related to different processes involved in converting a CFG to a proper CFG.

Properties for Initial Exclusion of Redundant Symbols

The properties for the first call of the algorithm for exclusion of redundant symbols include:

- Number of non-generating symbols
- Whether the language is empty
- Whether there is a rule with a right-hand side of non-generating symbols and terminals
- Number of unreachable non-terminals
- Number of unreachable terminals
- Whether there is a new unreachable symbol after exclusion of non-generating symbols

The *number of non-generating symbols* is the count of non-terminals removed from the grammar during the exclusion of non-generating symbols. The property *whether the language is empty* is set to true if the starting symbol is among the non-generating symbols. If true, we can terminate the process, as there will be no rules from the starting symbol, and after the exclusion of all unreachable symbols, the grammar will contain only the starting symbol without any non-terminals, terminals, or rules.

The property *whether there is a rule with a right-hand side of non-generating symbols and terminals* is there because it can make students misunderstand which non-terminals are generating and mistakenly mark a non-terminal on the left side of such a rule as generating.

Number of unreachable non-terminals and *number of unreachable terminals* track how many symbols are removed when excluding unreachable symbols. The property *whether there is a new unreachable symbol after exclusion of non-generating symbols* is crucial, as it identifies cases where a new unreachable symbol is created while excluding non-generating symbols. This helps track errors when students incorrectly reverse the order of algorithms, leading to an undesired grammar with remaining unreachable symbols.

Post-Redundant Symbols Removal Properties

The properties related to the grammar after exclusion of redundant symbols are:

- Number of non-terminals
- Number of terminals

Properties for The Exclusion of ε-rules

The properties for the process of removing ε-rules are:

- Number of ε-rules
- Whether the starting symbol generates ε
- Whether the starting symbol appears on the right side of any rule

Number of ε-rules is straightforward. The next two properties are set to `true` if we want the algorithm to create a new starting symbol with an ε-rule and a rule from the new symbol to the old starting symbol.

Properties for The Exclusion of Simple Rules

The properties for the process of removing simple rules are:

- Whether there are simple rules
- Whether there is transitivity over two simple rules
- Whether there is a new unreachable symbol after the exclusion of simple rules

We do not track the number of simple rules, as it is too complex to configure due to the many possibilities of the structure of the grammar after exclusion of ε-rules. Instead, we use a boolean value to indicate the presence of any simple rules. The property *whether there is transitivity over two simple rules* tracks if we find a simple rules closure for any non-terminal using at least two steps in the breadth-first search part of the algorithm. The last property, *whether there is a new unreachable symbol after the exclusion of simple rules*, identifies cases when there is a non-terminal reachable only by simple rules and becomes unreachable after their removal.

These properties are related to their respective algorithms: properties for ε-rules pertain to the grammar after excluding redundant rules, and properties for simple rules pertain to the grammar after excluding both redundant and ε-rules.

With all properties for the proper CFG problem established, we will now analyze the process of generating such a problem assignment.

4.3.6 Generation Process

To generate this problem assignment, we will consider all mentioned properties and utilize biased randomization.

We can split this generation into a three parts of a final CFG, the first part is generating a G_0, which will correpsond to a grammar after the exclusion of redundant symbols because when we will generate this grammar and then add a non-generating and unreachable symbols it will not change the G_0 at all because we will remove them and get the same G_0. So after the first part of generating G_0, we can add rules and generate a G_{-1}, which swill correspond to the grammar in the stage after first performing the algorithm for excluding all non-generating symbols. Finally, the G_{-2} will correspond to the input assignment itself, it will be the grammar in the stage before the whole conversion to the proper CFG. Another two stages of grammar G_1, G_2, up to G_4 will be performing exclusion of ε rules algorithm, exclusion of simple rules algorithm and again exclusion of non-generating and unreachable symbols. The diagram of stages of the grammar between algorithms, will be as follows:

The generation is divided into three stages for the final CFG. The first part is generating G_0, which corresponds to a grammar after exclusion of redundant symbols. Adding non-generating and unreachable symbols will not alter G_0, as they would be excluded to yield the same G_0. After generating G_0, we add rules to generate G_{-1}, corresponding to the grammar stage after exclusion of all non-generating symbols. Finally, G_2 corresponds to the input assignment itself, the grammar stage before converting to the proper CFG. The stages of the grammar G_1 through G_4 involve excluding ε-rules, simple rules, and again exclusion of non-generating and unreachable symbols. The diagram of these stages is shown in Figure 4.8.

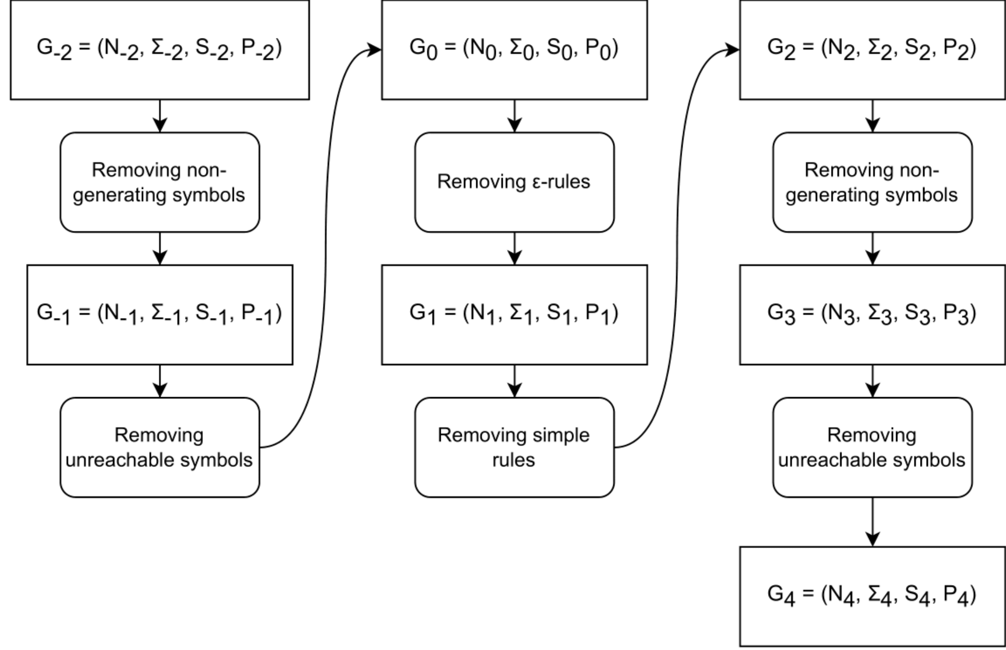


Figure 4.8: Stages of grammar during conversion to a proper cfg

We aim to avoid grammars where the right side of rules exceeds five symbols or contains many rules for one non-terminal. When generating a random rule for possible non-terminals on the left side, non-terminals with fewer production rules are chosen with higher probability. The statement "we will generate a rule of format $A \to \alpha$" includes an assumption that α is a maximum of five symbols long. If the right side is specified with a regular expression or a set of possible right sides, it is randomly chosen with some bias. However, five-symbol rules are rare, with most rules having 1 to 3 symbols on the right side.

Generating G_0

We start with generating $G_0 = (N_0, \Sigma_0, S_0, P_0)$. First, we set N_0 and Σ_0, determined by the *number of non-terminals* and *number of terminals* properties. The starting symbol S_0 is chosen randomly. Next, we create production rules for G_0.

To ensure every non-terminal symbol $A \in N_0$ is generating (as non-generating symbols are removed), we create a random permutation p_{N_0} of N_0:

$$p_{N_0} = \pi(N_0) = (A_1, A_2, \ldots, A_{|N_0|})$$

For the last k_s non-terminals from p_{N_0},

$$p_{N_0}[(|N_0| - k_s + 1) :] = (A_{|N_0|-k_s+1}, A_{|N_0|-k_s+2}, \ldots, A_{|N_0|}),$$

where k_s is the sum of $k_\varepsilon \leq |N_0|$ (the *number of ε-rules* property) and $k_{to_terminals} \in \{1, 2, \ldots, |N_0| - k_\varepsilon\}$. Each non-terminal $A \in p_{N_0}[(|N_0| - k_s + 1) :]$ has a rule of the format $A \to \Sigma_0^*$. In other words k_s is number of straightly generating symbols, symbols that we will be found in the first iteration of algorithm for

exclusion of non-generating symbols. The number $k_{to_terminals}$ is chosen randomly from 1 to $|N_0| - k_\varepsilon$ with an exponentially decreasing probability, avoiding too many straightly generating non-terminals.

After establishing p_{N_0} and $p_{N_0}[(|N_0| - k_s + 1) :]$, we generate rules of format $A \to \Sigma_0^+$ for the first $k_{to_terminals}$ non-terminals and $A \to \varepsilon$ for the remaining $k_\varepsilon = k_s - k_{to_terminals}$ non-terminals in $p_{N_0}[(|N_0| - k_s + 1) :]$. Then for each $A_i \in p_{N_0} \setminus p_{N_0}[(|N_0| - k_s + 1) :] = (A_1, A_2, \ldots, A_{|N_0|-k_s})$ we generate a rule:

- $A_i \to \alpha$, where $\alpha \in (p_{N_0}[i + 1 :] \cup \Sigma_0)^+$, with at least one symbol from $p_{N_0}[i+1 :]$ and Σ_0, if A_i is the starting symbol S_0 and the property *whether the starting symbol generates ε* is `false`, ensuring that the right side cannot be composed only of non-terminals generating ε, which would imply that the starting symbol also generates ε.

- $A_i \to \alpha$, where $\alpha \in (p_{N_0}[i + 1 :] \cup \Sigma_0)^+$, with at least one symbol from $p_{N_0}[i+1 :]$ and Σ_0, if the property *whether there are simple rules* is `false`, ensuring no simple rules are created while excluding ε-rules.

- $A_i \to \alpha$, where $\alpha \in (p_{N_0}[i + 1 :] \cup \Sigma_0)^+$, with at least one symbol from $p_{N_0}[i + 1 :]$, otherwise.

The sequence $p_{N_0}[(i + 1) :] = (A_{i+1}, A_{i+2}, \ldots, A_{|N_0|})$. This creates a directed acyclic graph (DAG) and ensures that each non-terminal is generating because each right side of a rule is made from generating non-terminals and terminals.

Before generating rules that ensure each non-terminal is generating, we must consider properties for the starting symbol and make adjustments because it could infringe some properties. If the property *whether the starting symbol is on the right side of any rule* is set to `false` or *whether the starting symbol generates ε* is `false`, we must swap the starting symbol in the p_{N_0} with the first symbol, so it cannot occur on the right side of any rule, preventing it from generating ε. Conversely, if the properties *whether the starting symbol generates ε* are `true` and *whether there are simple rules* are `false`, we must create a rule $S_0 \to \varepsilon$. If there is no rule $S_0 \to \varepsilon$, there must be a rule $S_0 \to C_1C_2 \ldots C_n$, where $C_i \in N_0$ and each C_i generates ε. Otherwise, S_0 would not generate ε, violating the first property. After performing the algorithm for exclusion of ε-rules, it will create new rules in the format $S_0 \to C_i$ for each $C_i \in \{C_1, C_2, \ldots, C_n\}$, creating simple rules and violating the second property. To prevent this, we force the addition of the rule $S_0 \to \varepsilon$. If the starting symbol is not in a position to add such a rule in the p_{N_0} (from $|N_0| - k_\varepsilon + 1$ to $|N_0|$), we set $k_\varepsilon \leftarrow k_\varepsilon - 1$. Then we can perform the generation of production rules as mentioned.

After ensuring that each non-terminal in G_0 is generating, we must ensure that each non-terminal is reachable so that we do not remove them while excluding unreachable symbols. First, we call an algorithm for exclusion of unreachable symbols on the so-far-generated G_0 and compare the final V_i, the set of reachable symbols, with N_0. If V_i includes all non-terminals from N_0, we are done. If not, we must add a few rules. We define U_0 as $U_0 = N_0 \setminus V_i$, all unreachable non-terminals. While $U_0 \neq \emptyset$, we iterate a process where we create a rule ensuring that at least one unreachable non-terminal from U_0 becomes reachable.

In each iteration, we add a rule for $A \in N_0$ in the form:

- $A \to \alpha$, where $\alpha \in (U_0 \cup \Sigma_0 \cup N_0)^+$, with at least one symbol from U_0 and Σ_0, if the property *whether there are simple rules* is `false` and the property *number of ε-rules* is not set to 0. We do not want to create rules with only non-terminals on the right side to prevent creating simple rules while excluding ε-rules. If the property *number of ε-rules* is set to 0, then while excluding ε-rules, nothing happens, so we can have right sides made up of only non-terminals.

- $A \to \alpha$, where $\alpha \in (U_0 \cup \Sigma_0 \cup N_0)^+$, with at least one symbol from U_0 and $|\alpha|$ is at least 2 symbols long if the property *whether there are simple rules* is set to `false`. We do not want to create a rule with only one non-terminal on the right side because it would be a simple rule.

- $A \to \alpha$, where $\alpha \in (U_0 \cup \Sigma_0 \cup N_0)^+$, with at least one symbol from U_0 otherwise.

After generating a new rule and adding it to P_0, we recalculate U_0, and if it is not empty, we go for another iteration.

So far, we have ensured that all non-terminals are generating and reachable. Now we will move into fulfilling other properties. The next property to be fulfilled is *whether the starting symbol is on the right side of any rule*. If the property is set to `true` and is not fulfilled yet, it could have been fulfilled in the process of generating rules for ensuring all non-terminals are generating and reachable. If it is not fulfilled yet, we generate a rule with S_0 on the right side. We will add a rule to P_0 for $A \in N_0$ in the form:

- $A \to \alpha$, where $\alpha \in (\Sigma_0 \cup \mathbb{N}_0)^+$ and there must be S_0 and at least one symbol from Σ_0 on the right side if the property *whether there are simple rules* is set to `false`. There must be at least one terminal because then in the removal of ε-rules, no new simple rules can be created from this rule.

- $A \to \alpha$, where $\alpha \in (\Sigma_0 \cup \mathbb{N}_0)^+$ and there must be S_0 on the right side otherwise.

The next property to check is *whether the starting symbol generates ε*. If it is set to `false`, we can skip to another part. If it is not, then we perform an algorithm for exclusion of ε-rules on the so-far-generated G_0 and check if S_0 is in N_ε from the algorithm process. If $S_0 \in N_\varepsilon$, generating ε, we can move on. If it is not, we must add a rule ensuring that it generates ε. We add a rule:

- $S_0 \to \varepsilon$, if the property *whether there are simple rules* is set to `false`. However, this option should never occur because this type of rule is added at the beginning of generating rules for G_0 to ensure that each non-terminal must be generating.

- $S_0 \to \alpha$, where $\alpha \in N_\varepsilon^+$. The right side is composed only of non-terminals generating ε, ensuring that S_0 does too.

If the property *whether there are simple rules* for G_1, meaning for a grammar after exclusion of ε-rules, is set to `false`, then we can skip to the next property. If it is not, we simplify a few generated rules for G_0 to avoid generating new

redundant rules, making the grammar too complex. If there are no rules for simplification, we must add new simple rules. Rules that can be simplified are in the format $A \to \alpha X \beta$, where $X, A \in N_0$ and $\alpha, \beta \in \Sigma_0^*$, meaning there is exactly one non-terminal on the right side. Simplifying this rule without disrupting already fulfilled properties is safe. Simplifying a rule with at least two non-terminals on the right side could break the invariant that all non-terminals in G_0 are reachable. Thus, we try to simplify any rule in the format $A \to \alpha X \beta$ to $A \to X$.

If we do not have a rule for simplification, we add random rules in the format:

- $A \to \alpha$, where $\alpha \in ((N_0 \cup N_\varepsilon) \setminus \{S_0\})^+$, with maximal one symbol from $(N_0 \setminus N_\varepsilon)$, if the property *whether the starting symbol is on the right side of any rule* is set to `false`. We avoid generating the starting symbol on the right side.
- $A \to \alpha$, where $\alpha \in (N_0 \cup N_\varepsilon)^+$, with maximal one symbol from $(N_0 \setminus N_\varepsilon)$ otherwise.

Where $A \in (N_0 - \{S_0\})$ if the property *whether the starting symbol generates* ε is set to `false`, else $A \in N_0$. Note that the probability of adding N_ε should not be high because there could be too many new rules during the ε-rules removal process. If $N_0 - \{S_0\} = \emptyset$, then we cannot create simple rules at all.

The next property, dependent on whether there are simple rules, is *whether there is a new unreachable symbol after exclusion of simple rules*. This property holds true when there exists a non-terminal that can only be reached using simple rules. To check this, we convert the grammar G_0 up to G_2 by first excluding ε rules, and then simple rules. G_2 undergoes an algorithm for exclusion of unreachable symbols, checking if $V_i \neq N_0$.

If the property is set to `true` and there can be simple rules at all, we will at the start of generating symbols G_0, move one non-starting symbol from generated non-terminals N_0 to a special non-terminal H. Then we do not work with this non-terminal in the process of generating so-far-generated G_0. Without checking if the property is already fulfilled, we generate two rules: the first, a simple rule with H on the right side, $A \to H$, where $A \in (N_0 - \{S_0\})$ if the *whether the starting symbol generates* ε property is `false`, else $A \in N_0$, to prevent the starting symbol from generating ε if H can generate ε; and the second rule ensuring H is generating in the form:

- $H \to \alpha$, where $\alpha \in ((N_0 \setminus \{S_0\}) \cup \Sigma_0)^+$, with at least one symbol from $(N_0 \setminus \{S_0\})$ if the property *whether the starting symbol is on the right side of any rule* is `false`.
- $H \to \alpha$, where $\alpha \in (N_0 \cup \Sigma_0)^+$, with at least one symbol from N_0 otherwise.

After generating rules for G_0 and these two rules for H, we add H to N_0. This ensures that if the property is `true`, it will always be fulfilled. We cannot prevent the situaton, that from previous generation it can end `true`, even it was set to `false` in the beginning.

The last property for simple rules, not so easy to fulfill, is *whether there is transitivity over two simple rules*. This property requires the existence of two rules $A \to B$ and $B \to C$ without a rule $A \to C$, where $A \neq B$, $A \neq C$, and

$B \neq C$. In other words, for at least one non-terminal, while exclusion of simple rules and finding a closure of simple rules, perform at least two iterations of a breadth-first search algorithm.

First, we check if the corresponding property is not yet fulfilled, assuming it is set to `true` and *whether there are simple rules* is also set to `true`. To check if it is fulfilled, we apply the simple rules removal algorithm on G_1, derived from G_0, and check if more than one iteration of breadth-first search is performed while finding closures of simple rules for each non-terminal. If not fulfilled, add a rule to extend a simple rule.

To add an extending rule for a simple rule in G_1, first we take the set of simple rules of G_1 over N_0, denoted by $SimpleRules_{P_1} = \{A \to B \mid A, B \in N_0, (A \to B) \in P_1, A \neq B\}$. We are taking only simple rules over N_0 because N_1 can be extended by a new starting symbol created while exclusion of ε-rules. Then select a random simple rule $X \to Y \in SimpleRules_{P_1}$ to extend.

We can attempt to extend it from the left if $X \neq S_0$ and the property *whether the starting symbol is on the right side of any rule* is set to `false`. If extending from the left, we add a rule in the format:

- $Z \to X$, where $Z \in (N_0 - \{S_0\}), Z \neq X, Z \neq Y$, if the property *whether the starting symbol generates* ε is `false` and the *number of* ε *rules* is not set to 0 to prevent X from generating ε and avoid $S_0 \to X$.
- $Z \to X$, where $Z \in N_0, Z \neq X, Z \neq Y$ otherwise.

Alternatively, we can try to extend it from the right if $Y \neq S_0$ and the property *whether the starting symbol generates* ε is `false` and the *number of* ε *rules* is not 0. If extending from the right, we add a rule:

- $Y \to Z$, where $Z \in (N_0 - \{S\}), Z \neq X, Z \neq Y$, if the property *whether the starting symbol is on the right side of any rule* is `true` to exclude S_0 from the right side.
- $Y \to Z$, where $Z \in N_0, Z \neq X, Z \neq Y$ otherwise.

If attempting to extend the rule $(X \to Y) \in SimpleRules_{P_1}$ fails, we remove $X \to Y$ and choose another rule from $SimpleRules_{P_1}$ to attempt extension. If all attempts fail, we assume this property cannot be fulfilled and end the G_0 generation process.

After generating the core grammar G_0, we wrap this grammar with additional unreachable symbols and their rules to obtain G_{-1}, and then generate non-generating non-terminals with rules to obtain the final product G_{-2}.

Generating G_{-1}

Assuming G_0 is successfully generated, we add additional unreachable symbols and rules to G_0 to obtain G_{-1}. Properties to fulfill include *number of unreachable non-terminals* and *number of unreachable terminals*.

We begin by generating non-terminals and terminals that will be unreachable. New unreachable non-terminals are denoted by N_{ur} and terminals by Σ_{ur}.

$|N_{ur}|$ corresponds to the *number of unreachable non-terminals*, and $|\Sigma_{ur}|$ corresponds to *number of unreachable terminals.* We ensure each unreachable symbol is generating and store all generated production rules in P_{ur}.

We randomly select $k_s \in \{1, 2, \ldots, |N_{ur}|\}$ with exponentially decreasing probability for larger values. This number determines the straightly generating non-terminals from N_{ur}. Then we create a random permutation

$$p_{N_{ur}} = \pi(N_{ur}) = (A_1, A_2, \ldots, A_{|N_{ur}|})$$

For the first k_s non-terminals $A \in p_{N_{ur}}[1 : k_s]$, we generate a rule:

- $A \to \alpha$, where $\alpha \in (N_0 \cup \Sigma_0 \cup \Sigma_{ur})^*$, with a low probability of getting the rule $A \to \varepsilon$.

For the remaining $|N_{ur}| - k_s$ non-terminals
A_i, where $i \in p_{N_{ur}}[k_s + 1, |N_{ur}|]$, which are not straightly generating, we create rules:

- $A_i \to \alpha$, where $\alpha \in (p_{N_{ur}}[1 : (i-1)] \cup \Sigma_{ur} \cup N_0 \cup \Sigma_0)^+$, with at least one symbol from $p_{N_{ur}}[1 : (i-1)]$.

We can see that the property *number of unreachable terminals* will not be always fullfiled but we do not need that. Unused terminals while generating rules for G_{-1}, we remove from Σ_{ur}. And we finally create $G_{-1} = (N_{-1}, \Sigma_{-1}, S_{-1}, P_{-1})$, where $N_{-1} = N_{ur} \cup N_0$, $\Sigma_{-1} = \Sigma_0 \cup \Sigma_{ur}$, $S_{-1} = S_0$, and $P_{-1} = P_0 \cup P_{ur}$. After that we can proceed to generate G_{-2}.

Generating G_{-2}

Assuming G_{-1} is successfully generated, we proceed to generate G_{-2}. We need to fulfill the properties of the redundant symbols part, including *number of non-generating symbols*, *whether there is a rule with the right side of non-generating symbols and terminals*, *whether there is a new unreachable symbol after exclusion of non-generating symbols*, and *whether the language is empty.*

We generate non-generating non-terminals N_{ng}, where $|N_{ng}|$ is the *number of non-generating symbols.* We use the set P_{ng} for new generated rules in G_{-2}.

To fulfill *whether there is a new unreachable symbol after exclusion of non-generating symbols*, we create paths. By creating a path $A_0 \to A_1 \to \ldots \to A_n$, we mean that we will create rules with the format of $A_0 \to \alpha_1 A_1 \beta_1$, $A_1 \to \alpha_2 A_2 \beta_2$, $\ldots$, up to $A_{n-1} \to \alpha_n A_n \beta_n$.

Our paths to create will be of the format:

$$A \to B_1 \to B_2 \to \ldots \to B_k \to C$$

Where $A \in N_0$, $B_1, B_2, \ldots, B_k \in N_{ng}$, and $C \in N_{ur}$,
we choose $k \in \{1, 2, \ldots, |N_0|\}$ randomly with decreasing probability for higher values. We avoid long paths to limit the number of rules. These paths ensure that non-terminals from N_{ur} at the end are reachable due to these paths if the algorithm for exclusion of unreachable symbols is performed first.

For each path $A \to B_1 \to B_2 \to \ldots \to B_k \to C$ and k, we generate a rule:

- $A \to \alpha$, where $\alpha \in (N_{ng} \cup \Sigma_{-1})^+$, with B_1 on the right side.

For each $i \in \{1, 2, \ldots, k\}$, we generate a rule:

- $B_i \to \beta$, where $\beta \in (N_{ng} \cup \Sigma_{-1} \cup N_0)^+$, with B_{i+1} on the right side, if $i < k$.
- $B_i \to \beta$, where $\beta \in (N_{ur} \cup N_{ng} \cup \Sigma_{-1} \cup N_0)$, with C and at least one symbol from N_{ng} on the right side if $i = k$.

The next property to fulfill is *whether there is a rule with the right side of non-generating symbols and terminals.* If it is set to `true`, we add rules of the format:

- $A \to \alpha$, where $A \in N_{ng}$, $\alpha \in ((N_{ng} \setminus \{A\}) \cup \Sigma_{-1})^+$, with at least one symbol from $(N_{ng} \setminus \{A\})$ and Σ_{-1} on the right side. We ensure the non-terminals on the right side do not equal to A, for making the possibility of mistakenly marking the A as generating.

Before moving to the last property, we add some additional random rules to make the grammar slightly more complex:

- $A \to \alpha$, where $A \in N_{ng}$, $\alpha \in (N_{ng} \cup N_{-1} \cup \Sigma_{-1})^+$, with at least one symbol from N_{ng} on the right side. This ensures that the rule will be removed during the exlusion of non-generating symbols and will not affect properties of generated grammar.

The last property, *whether the language is empty*, discredits all previous properties. If it is `true`, we want to make the starting symbol non-generating and then we will remove all rules during the first exclusion of non-generating symbols. To fulfill this, we set a non-generating non-terminal as the starting symbol while creating G_{-2}.

To create the final $G_{-2} = (N_{-2}, \Sigma_{-2}, S_{-2}, P_{-2})$, we set

- $N_{-2} \leftarrow N_{-1} \cup N_{ng}$,
- $\Sigma_{-2} \leftarrow \Sigma_{-1}$,
- $P_{-2} \leftarrow P_{-1} \cup P_{ng}$,
- $S_{-2} \leftarrow S_{-1}$.

If the *whether the language is empty* property is `true`, we set $S_{-2} \leftarrow A \in N_{ng}$.

Generation of Additional Rules

As the final step, we add a few random rules that do not affect the properties:

- $A \to \alpha$, where $A \in N_{ur}$, $\alpha \in (N_{-2} \cup \Sigma_{-2})^+$. This rule will be removed in the exclusion of unreachable symbols process.
- $A \to \alpha$, where $A \in N_0$, $\alpha \in (N_{ng} \cup N_{-1} \cup \Sigma_{-1})^+$, with at least one symbol from N_{ng} on the right side. This rule will be removed during the exclusion of non-generating symbols because it includes a non-generating symbol.

We add these rules to P_{-2}, and the process is complete. This finalizes the generation of a CFG for the *proper CFG* problem.

5. Documentation

The documentation for this project comprises three essential components: high-level programmer documentation, user documentation, and installation documentation. Each part serves a distinct purpose in ensuring comprehensive understanding and usability of the project.

5.1 High-Level Programmer Documentation

5.1.1 Formal Language Framework

The Formal Language Framework is a comprehensive library designed to represent objects from Formal Language theory. It includes modules for automata, grammars, and various utility functions. Let us begin with a high-level overview of its components and modules.

The framework consists of two main modules: *Grammars* and *Automata*.

Grammars

The classes in the *Grammars* module are:

- `Grammar` – Abstract base class for representing a grammar object, which inherits:
  - `CFG` – Base class for context-free grammar objects.
- `ProductionRule` – Represents a production rule with main properties: a body and a head, both tuple of symbols.
- `Symbol` – Abstract base class for symbols in grammar and production rules, with subclasses:
  - `NonTerminal` – Represents non-terminals.
  - `Terminal` – Represents terminals.
  - `Epsilon` – Represents an empty string, denoted by `$`.

Automata

The classes in the *Automata* module are:

- `Automaton` – Abstract base class for all automata objects.
- `FiniteAutomaton` – Abstract base class for finite automata, derived from `Automaton`.
  - `eNFA` – Represents ε-Non-Deterministic Automata, derived from `FiniteAutomaton`.
  - `NFA` – Represents Non-Deterministic Automata, derived from `eNFA`.
  - `DFA` – Represents Deterministic Automata, derived from `NFA`.

- `Symbol` – Represents symbols for the input alphabet of automata.
  - `Epsilon` – Special class derived from `Symbol`, represents an empty string.
- `State` – Represents a state in an automaton.
- `TransitionFunction` – Represents a transition function δ in an automaton, using `State` and `Symbol` functions.

The Formal Language Framework also includes a base abstract class `FormalLanguageObject` for each object, such as grammars or automata. The `Grammar` and `Automaton` abstract classes inherit from it. The entire inheritance hierarchy is shown in Figure 5.1.

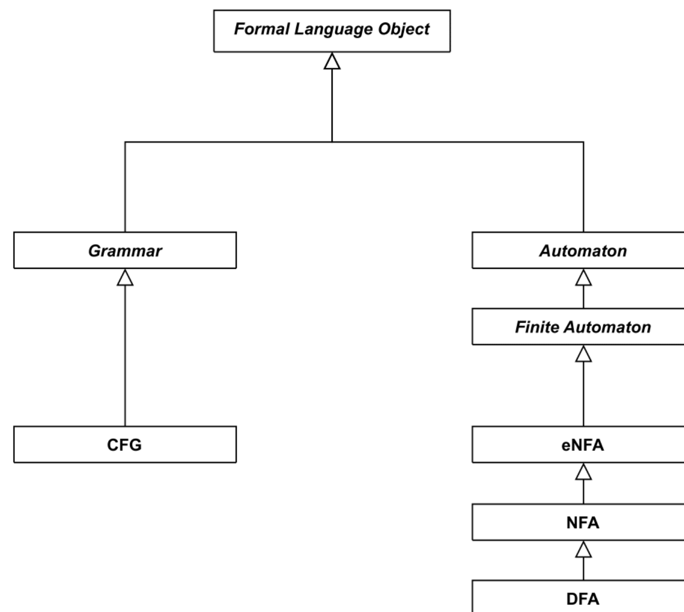


Figure 5.1: Formal Language Framework Objects Hierarchy

5.1.2 Main Application

The main part of the application is the `JobHandler` class. It parses attributes, loads the job file, and initializes `JobProperties` with properties from the job file to run the program. It also loads already generated base objects or decorated objects if specified in the attributes. The `JobHandler` then uses two classes, `JobCore` and `JobReporter`, to run the program. It first gives an instance of `JobProperties` to `JobCore` and executes it. `JobCore` handles the main logic, generating base objects, decorating them, and creating reports for each object based on `JobProperties` and additional settings. All generated objects and reports are stored within `JobCore`. After `JobCore` completes, it passes its instance to `JobReporter` for final reporting. `JobReporter` generates a HTML report from the given `JobCore`. Upon completion of `JobCore` and `JobReporter`, all generated base objects and their decorations from `JobCore` are serialized and stored in the

output folder. They are also converted into LaTeX format and stored in the same location. Additionally, the final report generated by `JobReporter` is saved as an `.html` file in the output folder.

`JobCore` uses `JobProperties` to drive the main logic of the program. It creates a generic instance of the `Problem` class based on the problem type using `ProblemFactory`. Then it initializes corresponding `ProblemProperties` and `DecoratorProperties` for each decoration from `JobProperties` (types defined by `Problem`). The sequence of operations in `JobCore` proceeds as follows: First, it generates a base object and creates a report for it by setting the required `ProblemProperties` to the `Problem`, executing `generate()` and then `report()` methods from the `Problem` instance, and storing the generated object and its report. Next, it creates decorations for the generated base object by setting the corresponding `DecorationsProperties` to the `Problem`, executing `decorate()` and `report()` methods of the `Problem`, and storing the decorated object and its report. This process repeats for the required number of generated objects.

The `Problem` abstract class serves as an abstraction over a problem. It contains a property, an instance of `ProblemObject`, corresponding to the problem input. The `Problem` class provides three main methods:

- `generate()` – Generates an object of `ProblemObject` corresponding to the input and stores it in the input object property.
- `decorate()` – Decorates the input object.
- `report()` – Takes the input object, solves it, creates a report from it, and stores the report in the property.

The primary objective of the `Problem` class is to integrate corresponding types of abstract classes: `ProblemGenerator`, `ProblemObjectDecorator`, and `ProblemSolver`. The `ProblemGenerator` is used in the `generate()` method to generate a problem object based on `ProblemProperties` (configurable properties for the problem). The `ProblemObjectDecorator` class decorates the corresponding `ProblemObject` (input of the problem) in the `decorate()` method. Finally, the `ProblemSolver` class solves the problem using the input object, creates a report, and tracks problem properties.

The classes `ProblemGenerator`, `ProblemSolver`, `ProblemProperties`, and `Problem` are abstract classes, with non-abstract derived classes created for each problem type supported. Likewise, classes `ProblemObject`, `ProblemObjectDecorator`, and `ProblemObjectDecoratorProperties` are abstract classes, with non-abstract derived classes created for each type of input object for supported problem types.

Note that the `ProblemObject` class often comprises `FormalLanguageObject` objects, and each class representing properties (e.g., `JobProperties`, `ProblemProperties`, or `ProblemObjectDecoratorProperties`) derives from the abstract class `Properties`.

Figure 5.2 illustrates the relationships and class usage across the entire program.

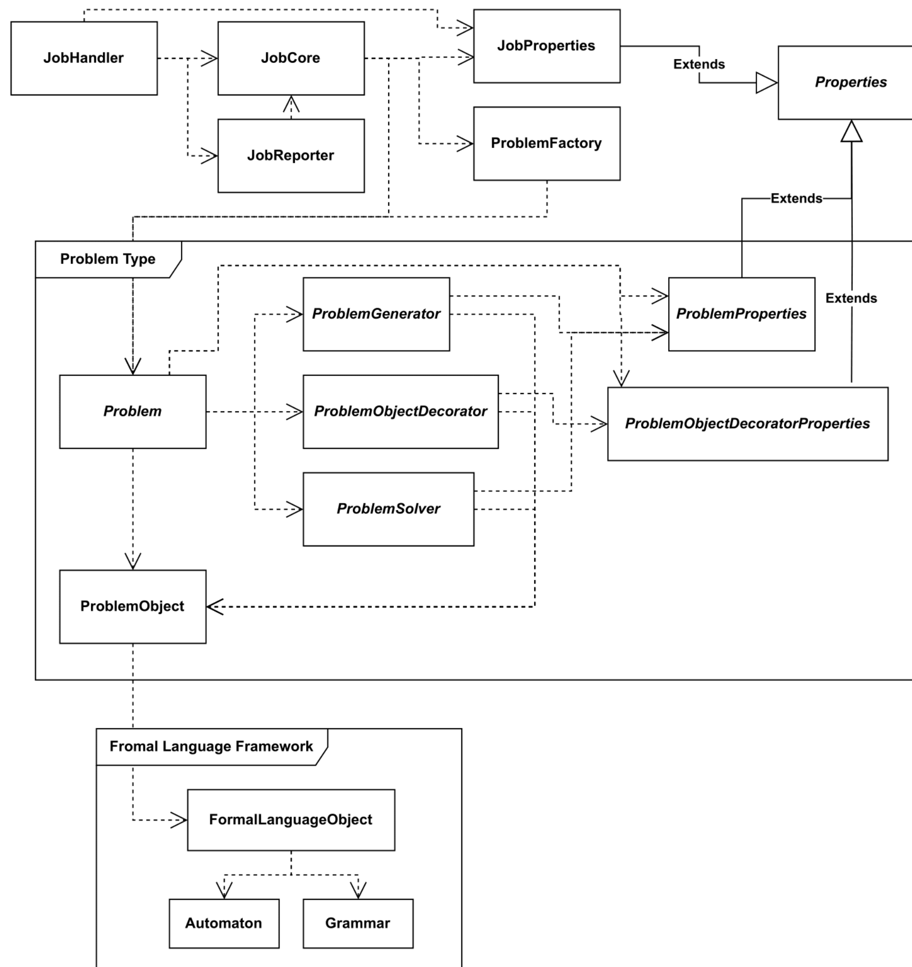


Figure 5.2: Relations between classes in the application

5.2 Testing

We have included automated tests for various components of the program.

Initially, tests ensure that objects from the Formal Language Framework accurately represent formal language theory objects. Subsequently, tests validate core functionalities of the main application.

For each `ProblemSolver`, tests verify the correctness of the solving algorithm, ensuring accurate solutions and property tracking. Tests for each `ProblemObjectDecorator` ascertain proper decoration of the problem object with specific decorator properties.

The `ProblemGenerator` undergoes testing with diverse property values to ensure it yields reasonable outcomes, as evidenced in the attachments in the `examples` directory.

Furthermore, we have tested the robustness of our application against user errors, such as setting properties with nonsensical values. In such cases, the application identifies the erroneous property, preventing job execution. The application also handles errors during object generation, decoration, and report

creation gracefully, terminating without exceptions in case of failure.

5.3 User Documentation

This program is an application for generating problem assignments based on Formal Language, with configurable properties.

To run the program, download the repository and execute the script *run.bat* on Windows or *run.sh* on Unix-like systems (Linux, macOS).

Each program run requires specifying the path to a job file. The job file is a JSON configuration file containing settings for program execution, configurable properties for the problem, and configurable properties for each decoration of the generated object. The configurable properties for the problem are used to generate base objects, where each corresponds to a single problem assignment. The configurable properties for each decoration, by that we mean a variation, with different naming of the problem assignment, are used to create decorations for these base objects.

After running the program, a directory named after the job file with a `_output` suffix will be created in the same path. This directory will store:

- Generated base objects in serialized and LaTeX formats.
- Decorations of base objects for each problem assignment in serialized and LaTeX formats.
- The final report in HTML format.

Let us walk through an example run of the program for the "Determinization of NFA" problem, assuming Windows OS.

1. Create a job configuration file to define program settings:

   ```
   >>> run.bat job_file.json --CreateConfig
   ```

   This command prompts you to specify a problem type after the `--CreateConfig` attribute and lists available problem types. For instance, choosing `nfa_to_dfa`:

   ```
   >>> run.bat job_file.json --CreateConfig nfa_to_dfa
   ```

   This generates the configuration file.

2. Modify the configuration file `job_file.json` as needed.

3. Execute the command to run the program and generate problem assignments:

   ```
   >>> run.bat job_file.json
   ```

This command runs the program with default behavior, generating base objects, creating decorations, generating LaTeX files for problem assignments and a final summary report. All output is stored in the `job_file_output` directory.

A configuration file for the `nfa_to_dfa` problem looks like this:

```
{
    "problem_type": "nfa_to_dfa",
    "problems_count": 10,
    "decorations_count": 5,
    "decorations_properties": [
        ...
    ],
    "one_file_problems": false,
    "one_file_decorations": false,
    "one_file_latex": false,
    "problem_properties": {
        ...
    }
}
```

Configuration File Properties

- `problem_type`: Specifies the type of problem.
- `problems_count`: Number of problem assignments (base objects) to generate.
- `decorations_count`: Number of decorations to create for each problem assignment.
- `decorations_properties`: List of properties used for creating decorations, with length corresponding to `decorations_count`.
- `one_file_problems`: If `true`, stores all generated base objects in a single file.
- `one_file_decorations`: If `true`, stores all decorations in a single file.
- `one_file_latex`: If `true`, stores all LaTeX versions of base objects in a single file.
- `problem_properties`: Required properties for generating base objects.

Problem Properties for `nfa_to_dfa` Problem

- `alphabet_length`: Number of symbols in the alphabet for the generated NFA.
- `states_count`: Number of states in the generated NFA.

- `initial_states_count`: Number of initial states.
- `final_states_count`: Number of final states.

Problem Properties for `cyk` Problem

- `input_string_length`: Length of the input string for checking acceptance.
- `terminals_count`: Number of terminals in the CFG.
- `non_terminals_count`: Number of non-terminals in the CFG.
- `starting_symbol_generating_epsilon`: Equals `true` if the CFG includes the rule $S \to \varepsilon$.

Problem Properties for `proper_cfg` Problem

- `non_generating_count`: Number of non-generating symbols to remove initially.
- `empty_language`: Equals `true` if the grammar generates an empty language.
- `right_side_from_non_generating_and_terminals`: Equals `true` if there is a rule with a right side consisting of non-generating symbols and terminals.
- `unreachable_non_terminals_count`: Number of unreachable non-terminals found during the initial removal of redundant symbols.
- `unreachable_terminals_count`: Number of unreachable terminals found during the initial removal of redundant symbols.
- `new_unreachable_after_exclusion_non_generating`: Equals `true` if new unreachable symbols are found after excluding non-generating symbols.
- `non_terminals_count`: Number of non-terminals after excluding redundant symbols.
- `terminals_count`: Number of terminals after excluding redundant symbols.
- `epsilon_rules_count`: Number of ε-rules after excluding redundant symbols.
- `starting_symbol_generating_epsilon`: Equals `true` if the starting symbol generates ε after excluding redundant symbols.
- `starting_symbol_on_right_side`: Equals `true` if the starting symbol appears on the right side of any rule after excluding redundant symbols.
- `simple_rules`: Equals `true` if there are any simple rules after excluding redundant symbols and ε-rules.
- `transitivity_over_two_simple_rules`: Equals `true` if at least two iterations are required to find a closure over simple rules.

- `new_unreachable_after_exclusion_simple_rules`: Equals `true` if new unreachable symbols are found after excluding simple rules.

Note that generated base objects attempt to match these attributes but may not exactly replicate them.

Decorations Properties for an Automaton (`nfa_to_dfa` Problem)

- `states`: String specifying states for the property, with each character representing a state. If insufficient number of states is specified, new states are created using the last character plus an index, starting from 0.
- `input_alphabet`: String specifying symbols for the new input alphabet, starting from the first character. If insufficient number of symbols is specified, the decoration is not created.

Decorations Properties for a Grammar (`cyk` and `proper_cfg` Problems)

- `non-terminals`: String specifying new non-terminals, starting from the first character. The first character of the string is used for the starting symbol.
- `terminals`: String specifying new terminals, starting from the first character.

If insufficient symbols are specified for non-terminals or terminals, the decoration is not created.

Optional Attributes

Optional attributes control program phases depending on previous runs:

- `--CreateConfig PROBLEM_TYPE`: Generates a configuration file for the first program run, specific to the `PROBLEM_TYPE` problem.
- `--Generate`: Generates new base objects based on problem properties in the job file.
- `--Decorate`: Creates new decorations for each base object based on properties in the job file.
- `--Latex`: Generates LaTeX files for each base object and its decorations.
- `--Report`: Generates a final report based on the generated and decorated objects.

By default, if no specific attributes are specified, the program runs equivalently to:

```
>>> run.bat JOB_FILE --Generate --Decorate --Latex --Report
```

or simply:

```
>>> run.bat JOB_FILE
```

If run without `--Generate`, previously generated base objects are loaded. If run without `--Generate` and `--Decorate`, previously generated and decorated objects are loaded. These assumptions apply when running the program with the same settings after at least one prior execution. Omitting `--Decorate` and including `--Generate` skips decoration creation. Omitting `--Latex` or `--Report` skips LaTeX file creation or report generation.

Additional attributes include:

- `--ReportFile REPORT_FILE`: Specifies a custom name or path for the report file.
- `--OutputFolder OUTPUT_FOLDER`: Specifies a custom name and path for the output folder, where all outputs are stored or loaded from.

5.4 Instalation Documentation

The only requirement for running this program is to have Python 3.0 installed [24]. This program does not require any additional installation steps. Simply download the repository and execute the script *run.bat* on Windows or *run.sh* on Unix-like systems (Linux, macOS).

Conclusion

In the beginning, we found that there are no existing generators of objects or problem assignments for formal language theory. Because of this, we investigated existing formal language libraries and frameworks and their capabilities. Based on this knowledge, we specified requirements and created our own formal language framework, which we use in our final generator.

We then analyzed and specified our desired application for generating problem assignments. We proposed the structure of such a generator and successfully implemented it.

Our application supports two simpler problems: the *determinization of NFA to DFA* and the *CYK problem*, which checks if a string is generated by a corresponding CFG. Additionally, it handles one complex problem, the *proper CFG* problem, which involves converting a CFG to a proper CFG. This problem includes several subproblems, such as the exclusion of redundant symbols, the exclusion of ε-rules, and the exclusion of simple rules.

For each problem, we can specify the required properties or situations that we want to test with the generated assignment. We can then create variations with different namings of states, symbols, etc., for each problem assignment. Another utility of the application is that it provides a final report with the generated assignments and their variations. For each assignment, it compares the actual properties with the required ones and includes the entire process of solving the problem along with the final solution. Finally, it converts each problem assignment into the LaTeX format for easy inclusion in tests created for students.

One problem we faced was how to configure the running of the program and the required properties for the generated assignments and their variations. We solved this with JSON configuration files with nested properties. The main JSON configuration file includes the main settings for running the program, a list of required properties for each decoration, and a field with nested required properties for generated problem assignments. For each type of property, we created a base class *Properties*, ensuring that each properties object can be easily serialized into and from JSON.

Another complication was with the *proper CFG* problem, where we had to address the complexity of the problem and its composition of several subproblems. The generation process is not trivial and includes several phases of generation. The next challenge was how to use the composition of subproblems to our advantage and create a solver that solves the problem, generates a report from the solution process, and tracks the properties for the entire problem. We implemented smaller subsolvers to solve each subproblem and track the relevant properties. The solver for the *proper CFG* problem is composed of these subsolvers and tracks additional properties relevant to the whole process.

Another utility we wanted to fulfill was the ability to change the generated assignments since a slight change can sometimes yield a reasonable assignment. We achieved this by adding options to skip the generation phase and saving the generated assignments into files. Since we added the serialization of generated assignments or input objects of the problem, we can generate them, modify them in their serialized form in the files, and then restart the program, skipping the

generation phase and loading and deserializing them from the files.

Lastly, we encountered the non-functional requirement that our application must be extensible to include other problems. To address this, we established an abstraction over problems in the `Problem` class, where we specify the input object of the problem, the corresponding generator, decorator (for creating variations), solver, etc. Thanks to this abstraction, we can even add problems not only from formal language theory.

Contributions

Our main contributions are:

- A formal language framework that supports objects from automata and grammars theory.
- An application for generating problem assignments with required properties from formal language theory.
- The application includes the creation of variations for each problem assignment.
- The application includes a report from the solving process and the final solution.
- Three supported problems: the determinization, the CYK problem and the proper CFG problem.
- Extensibility of the application for other problems, not limited to formal language theory.

Future Work

The main future work lies in adding new problems, which includes implementing the generator, decorator, solver, and other necessary components.

Another area for improvement is creating IDs for exceptions raised during the run of the program and generating better output to the console during the execution process of the application.

Bibliography

[1] Manish Bhojasia. Automata Theory MCQ (Multiple Choice Questions). https://www.sanfoundry.com/1000-automata-theory-questions-answers/. Accessed on: 2023–09–05.

[2] Christopher Wong. CFG Developer. https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/, 2014. Accessed on: 2023–09–05.

[3] Wikipedia contributors. Earley parser. https://en.wikipedia.org/w/index.php?title=Earley_parser&oldid=1155305622, 2023. Accessed on: 2023–09–05.

[4] Jonathan Hall. Question Generator. https://mathsbot.com/questionGenerator. Accessed on: 2023–09–05.

[5] Microsoft Corporation. Microsoft onenote. https://products.office.com/onenote. Accessed on: 2023–09–05.

[6] Ramsri Goutham Golla. Questgen. https://www.questgen.ai/ai-similar-question-generator. Accessed on: 2023–09–05.

[7] Jonathan Goldman. Algorithmically Generating Questions. https://medium.com/knerd/algorithmically-generating-questions-a786b5fb0a15, Year of publication: 2013–09–19. Accessed on: 2023–09–05.

[8] Sumit Gulwani, Erik Andersen, and Zoran Popović. A Trace-based Framework for Analyzing and Synthesizing Educational Progressions. In *CHI 2013, April 27-May 2, 2013, Paris, France*, April 2013.

[9] Julien Romero. Pyformlang: An Educational Library for Formal Language Manipulation, 2021.

[10] Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. Exploring network structure, dynamics, and function using networkx. 2008.

[11] Susan H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., USA, 2006.

[12] Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 4 1991.

[13] T. Boutell. PNG (Portable Network Graphics) Specification Version 1.0. Technical report, 3 1997.

[14] Malte Isberner and Markus Frohme. Automatalib: A java library for data structures and algorithms for automata. https://github.com/LearnLib/automatalib, 2015. Accessed on: 2023–08–25.

[15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. Ltsmin: High-performance language-independent model checking. In *TACAS 2015*, 2015.

[16] Jan Trávníček, Tomáš Pecka, and Štěpán Plachý. Algorithms Library Toolkit. `https://gitlab.fit.cvut.cz/algorithms-library-toolkit`, 2019. Accessed: 2023–08–25.

[17] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, oct 1964.

[18] V. M. Glushkov. On formal transformations of algorithms. In Andrei P. Ershov and Donald E. Knuth, editors, *Algorithms in Modern Mathematics and Computer Science*, pages 430–440, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[19] Guangming Xing. Minimized thompson nfa. *International Journal of Computer Mathematics*, 81(9):1097–1106, 2004.

[20] Ivan Zuzak and Vedrana Jankovic. Web application regular expressions gym. `https://ivanzuzak.info/noam/webapps/regex_simplifier/`, 2024. Accessed on: 2024–05–25.

[21] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON Schema. Technical report, 4 2016.

[22] T. Berners-Lee and D. Connolly. Hypertext Markup Language - 2.0. Technical report, 11 1995.

[23] Hakon Wium Lie and Bert Bos. *Cascading Style Sheets: Designing for the web*. 4 1997.

[24] Python Software Foundation. Python 3.0 Documentation, 2008.

List of Figures

1.1 Visualization of DFA using Pyformlang 10
1.2 Visualization of minimized DFA using Pyformlang 10

3.1 Formal Language Framework modules 30
3.2 Grammars module classes 31
3.3 Automata module classes 32
3.4 Formal Language Framework objects hierarchy 33
3.5 Problem Object hierarchy 35
3.6 Problem Generator classes 36
3.7 Problem Decorator classes 36
3.8 Problem Solver classes 37
3.9 Problem classes 38
3.10 Problem report design 39

4.1 Algorithm for determinization of NFA 42
4.2 CYK Algorithm 46
4.3 Algorithm for exclusion of non-generating symbols from CFG . . . 50
4.4 Algorithm for exclusion of unreachable symbols from CFG 51
4.5 Algorithm for exclusion of redundant symbols from CFG 51
4.6 Algorithm for exclusion of ε-rules from CFG 52
4.7 Algorithm for exclusion of simple rules from CFG 53
4.8 Stages of grammar during conversion to a proper cfg 57

5.1 Formal Language Framework Objects Hierarchy 65
5.2 Relations between classes in the application 67

List of Tables

1.1 General information about libraries. 17
1.2 External information about libraries. 18
1.3 Capabilities of libraries in regular languages. 18
1.4 Capabilities of libraries in context-free languages. 19

A. Attachments

A.1 `generator_application.zip`

- `README.md` – A file containing basic information about the program.
- `run.bat` – A script for running the application on the Windows platform.
- `run.sh` – A script for running the application on the Linux platform.
- `app/` – A directory containing the source code of the application.
- `app/docs/` – A directory containing the generated documentation.
- `jobs_examples/` – A directory containing examples of generated problem assessments.