NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO BERNARDO GUERREIRO DOS SANTOS

BSc in Computer Science and Engineering

FINITE-STATE TRANSDUCERS IN OCAMLFLAT/OFLAT

Dissertation Plan

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

July, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

FINITE-STATE TRANSDUCERS IN OCAMLFLAT/OFLAT

JOÃO BERNARDO GUERREIRO DOS SANTOS

BSc in Computer Science and Engineering

Adviser: Artur Miguel Dias
Assistant Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2025

Abstract

This thesis presents the design and implementation of finite state transducers (FSTs) within the OCamlFLAT/OFLAT application, a web-based educational tool for exploring concepts in computation theory developed at FCT NOVA. Finite state transducers extend finite automata by associating output symbols with transitions, enabling the modeling of transformations from input to output strings.

The current system consists of two separate components: Firstly we have the OCamlFLAT library, a library with all the data structures and logic regarding the theoretical concepts of its supported models. And secondly the OFLAT web application, which is built on top of the OCamlFLAT library, provides a pedagogical interactive graphic interface for users to explore the models supported by the library. Both tools were written in OCaml, deliberately aiming to use a functional declarative style, which in some cases manages to closely resemble the theoretical formalisms taught to students. It is a system that is in constant evolution, with new features being added to both the library and the web application.

The work to be developed in this thesis involves the extension of these tools with new operations over transducers, such as translation, acceptance, generation, classification and conversion. Additionally, the new concepts must be integrated in OFLAT's ecosystem, which includes model composition, and pedagogical exercises for students to practice and test their knowledge.

The resulting system will allow users to build, modify, and analyze FSTs visually, supporting better comprehension of theoretical principles through experimentation.

Resumo

Esta dissertação apresenta o projeto e a implementação de transdutores finitos (FSTs) na aplicação OCamlFLAT/OFLAT, uma ferramenta educacional baseada na web para a exploração de conceitos de teoria da computação, desenvolvida na FCT NOVA. Os transdutores finitos estendem os autómatos finitos ao associar símbolos de saída às transições, permitindo a modelação de transformações entre cadeias de entrada e de saída.

O sistema atual é composto por dois componentes distintos: a biblioteca OCamlFLAT, que contém todas as estruturas de dados e lógica associadas aos conceitos teóricos dos modelos suportados; e a aplicação web OFLAT, construída sobre a biblioteca OCamlFLAT, que fornece uma interface gráfica interativa com fins pedagógicos para que os utilizadores possam explorar os modelos suportados pela biblioteca. Ambas as ferramentas foram desenvolvidas em OCaml, procurando adotar deliberadamente um estilo funcional declarativo, que em alguns casos se aproxima dos formalismos teóricos ensinados aos alunos. Trata-se de um sistema em constante evolução, com novas funcionalidades a serem acrescentadas continuamente tanto à biblioteca como à aplicação web.

O trabalho a desenvolver nesta dissertação envolve a extensão destas ferramentas com novas operações sobre transdutores, tais como tradução, aceitação, geração, classificação e conversão. Adicionalmente, os novos conceitos devem ser integrados no ecossistema do OFLAT, o que inclui a composição de modelos e a criação de exercícios pedagógicos para os alunos praticarem e testarem os seus conhecimentos.

O sistema resultante permitirá aos utilizadores construir, modificar e analisar transdutores finitos de forma visual, promovendo uma melhor compreensão dos princípios teóricos através da experimentação.

Contents

List of Figures

Listings

1 Introduction

1.1 Context

Theory of Computation is a fundamental area in computer science that includes various fields of study, including this work's focus: Formal Languages and Automata Theory (FLAT). These play a critical role in modeling computational processes, verifying system behavior, and designing compilers and interpreters.

Despite its theoretical significance, FLAT concepts can often be difficult to understand due to their abstract nature. Especially for students new to the field, to address this, educational tools for visualization and simulation of these concepts are an invaluable resource. Among these tools, OCamlFLAT/OFLAT, a system developed at NOVA School of Science and Technology, provides a comprehensive platform for exploring various models of computation, including finite automata, grammars, pushdown automata, and Turing machines. However despite its extensive support for various models, OCamlFLAT/OFLAT currently lacks support for finite-state transducers.

Finite-State Transducers (FSTs) are an extension of the concept of finite automata, as they introduce output generation in response to input processing. And while FSTs are well-established in theoretical literature, pedagogical tools that allow for their interactive construction, simulation, and analysis remain scarce or limited in functionality.

This dissertation aims to address this issue by extending both the library and the web interface to support the representation, manipulation, and simulation of FSTs. The goal being that this addition will make OCamlFLAT/OFLAT an even more complete educational tool for students and educators alike.

1.2 Objectives

The primary goal of this thesis is to design and implement support for finite-state transducers within the OCamlFLAT/OFLAT ecosystem. This includes:

- Defining appropriate data structures and logic to model FSTs.

- Implementing key operations such as word translation, language recognition, determinization, and minimization.
- Integrating FSTs into the OFLAT graphical interface with interactive features for construction, simulation, and step-by-step word processing.
- Ensuring consistency with existing models, including support for composition and pedagogical exercises.

By enriching the system with transducer functionality, we aim to provide a more comprehensive pedagogical platform that supports a broader spectrum of the Chomsky hierarchy and theoretical computation models. This will enhance both teaching and learning by enabling students and instructors to explore complex language transformations through hands-on experimentation.

1.3 Structure of the Document

The remainder of this document is structured as follows.

- Chapter 2 presents the theoretical foundations of formal languages and automata theory, focusing on finite-state transducers and their operations.
- Chapter 3 reviews existing related tools for simulation of various models and compares them to the OCamlFLAT/OFLAT system.
- Chapter 4 delves into the current functionalities and user experience of OCamlFLAT/OFLAT.
- Chapter 5 describes the technologies and frameworks used in the development of the OCamlFLAT/OFLAT system, including the OCaml programming language and the OFLAT web interface
- Chapter 6 outlines the detailed work plan for extending the system with FST support.

2

Background

2.1 Introduction

The work done within the OFLAT application uses a model from the Theory of Computation, the concepts behind Finite-State Transducers (FSTs) need to be explained beforehand so that the rest of the document can be understood with ease. Starting from the higher-level concepts first and going down to the Finite-State Transducers which is what we will focus on for the purpose of this document.

2.1.1 Formal Language and Automata Theory

Formal Language and Automata Theory (referred to as FLAT from here onwards), is technically a composition of two subjects, Formal Language Theory and Automata Theory that, while they are different subjects, they are related and often studied and mentioned together. Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them. An automaton (automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically. An automaton with a finite number of states is called a finite automaton (FA) or finite-state machine (FSM).

A formal language is a set of strings whose symbols are taken from a set called "alphabet". The alphabet of a formal language consists of symbols that concatenate into strings (also called "words"). A formal language is often defined by means of a formal grammar such as a regular grammar or context-free grammar. In computer science, formal languages are used, among others, as the basis for defining the grammar of programming languages and formalized versions of subsets of natural languages, in which the words of the language represent concepts that are associated with meanings or semantics.

In FLAT, automata are used as finite representations of formal languages that may be infinite. Automata are often classified by the class of formal languages they can recognize, as in the Chomsky hierarchy(we will see later), which describes a nesting relationship between major classes of automata.

2.2 Languages and Grammars

Languages can be expressed over two different methods: Recognition (via the use of automata) and Generation (via the use of grammars). In generation a language is represented by a grammar and is a set of all the words that can be derived from the start symbol of the grammar. While in recognition, a language is represented by an acceptor (usually an automaton) and is a set of all the words that are accepted by the automaton. Finite automata and FSTs are recognizers so our focus will be on them, although with FSTs there is more nuance to that concept, as we will see later.

2.2.1 Chomsky Language Hierarchy

The Chomsky hierarchy[15] is a containment hierarchy of classes of formal grammars. Noam Chomsky theorized that four different classes of formal grammars existed that could generate increasingly complex languages. Each class can also completely generate the language of all inferior classes (set inclusive). The general idea of a hierarchy of grammars was first described by Noam Chomsky in "Three models for the description of language". Marcel-Paul Schützenberger also played a role in the development of the theory of formal languages; the paper "The algebraic theory of context free languages" describes the modern hierarchy, including context-free grammars. Independently, alongside linguists, mathematicians were developing models of computation (via automata). Parsing a sentence in a language is similar to computation, and the grammars described by Chomsky proved to both resemble and be equivalent in computational power to various machine models. The Chomsky hierarchy is often represented as follows:

Type 0: Recursively enumerable languages, recognized by Turing machines. These include all problems that can be algorithmically solved.

Type 1: Context-sensitive languages, recognized by linear bounded automata.

Type 2: Context-free languages, generated by context-free grammars and recognized by pushdown automata. Common in programming languages and syntactic structures.

Type 3: Regular languages, the simplest class, generated by regular grammars and recognized by finite automata. They represent patterns describable by regular expressions.

Finite-State Transducers operate at the Type 3 level, dealing with regular languages.

2.2.2 Regular Languages

Regular languages are a class of formal languages that can be defined by a regular expression and recognized by finite automata. They represent the simplest type of language in the Chomsky hierarchy (Type 3). A language L is said to be regular if there exists a finite automaton that accepts exactly the strings in L.

Regular languages are defined semantically over an alphabet Σ as follows:

- The empty language $\emptyset$ is a regular language.

- For each a ∈ Σ (a belongs to Σ), the singleton language a is a regular language.
- If A is a regular language, A* (Kleene star) is a regular language. Due to this, the empty string language Σ is also regular.
- If A and B are regular languages, then A ∪ B (union) and A B (concatenation) are regular languages.
- No other languages over Σ are regular.

2.2.3 Regular Expressions

Regular expressions are a common way to describe a language in a practical way, it consists of constants, which denote sets of strings, and operator symbols, which then denote operations over these sets. The following definition is standard, and found as such in most textbooks on formal language theory. Given an alphabet Σ, regular expressions are built from the following components:

- $\emptyset$ and Σ are regular expressions.
- Every element in Σ is a regular expression.
- Concatenation of two regular expressions α and β, produces a regular expression α β
- Union of two regular expressions α and β, produces a regular expression $\alpha \mid \beta$
- Kleene star of α is a regular expression.

The following illustrates a regular expression denoting a language:

$$
\begin{aligned}
L(\emptyset) &= \emptyset, \\
L(\lambda) &= \{\lambda\}, \\
L(a) &= \{a\} \quad (a \in T), \\
L(E + F) &= L(E) \cup L(F), \\
L(EF) &= L(E)L(F), \\
L(E^*) &= L(E)^*.
\end{aligned}
$$

For example, the regular expression $(ab)^* + c$ describes a language that consists of any number of repetitions of the string "ab" followed by the string "c".

$$\begin{aligned} L((ab)^* + c) &= L((ab)^*) \cup L(c) \\ &= L(ab)^* \cup \{c\} \\ &= [L(a)L(b)]^* \cup \{c\} \\ &= [\{a\}\{b\}]^* \cup \{c\} \\ &= \{ab\}^* \cup \{c\}. \end{aligned}$$

2.3 Finite Automata

Finite automata are among the most basic computational models in automata theory. They consist of a finite number of states and transitions between those states based on input symbols. They are capable of recognizing regular languages. A key concept in understanding how finite automata process input is the notion of a configuration. A configuration represents the current status of the automaton at a particular point during the computation. It is typically defined as a 2-tuple (q,w), where q is the current state and w is the remaining input word. As the automaton processes the input, it transitions from one configuration to another based on the current symbol and the transition function. The input is said to be accepted if at least one of these paths reaches a final state with no remaining input. Any finite automaton can be made deterministic, and minimized. We will explore how minimization works later on in the FST section. There are two types of finite automata:

- **Deterministic Finite Automata (DFA)**: For each state and input symbol, there is exactly one transition.
- **Nondeterministic Finite Automata (NFA)**: Multiple transitions may be possible for the same input symbol in a given state.

2.3.1 Deterministic Finite Automata (DFAs)

Deterministic Finite Automata (DFA) are a special case of finite automata where, for each state and input symbol, there is exactly one transition to a next state. So there are no λ-transitions and no state has more than one transition for each input symbol. This determinism simplifies the computation process, as there is no ambiguity in state transitions. A deterministic finite automaton is formally defined as a 5-tuple [5]:

$$M = (Q, \Sigma, \delta, q_0, F),$$

where:

- Q is a finite set of states,
- Σ is the input alphabet,

- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final (accepting) states.

The DFA operates as follows: it begins in the initial state q_0, with the input mechanism positioned at the leftmost symbol of the input string. At each computational step, the automaton reads one input symbol, transitions to a new state as defined by δ, and advances the input mechanism one position to the right. This process continues until the entire input string has been consumed. If, at the end of the input, the automaton is in a state belonging to F, the input is accepted; otherwise, it is rejected.

The transition function δ dictates state changes. For example, if

$$\delta(q_0, a) = q_1,$$

then when the automaton is in state q_0 and reads the symbol a, it transitions to state q_1.

Consider the DFA defined by:

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

where the transition function δ is represented in the transition graph shown in Figure 1.2.

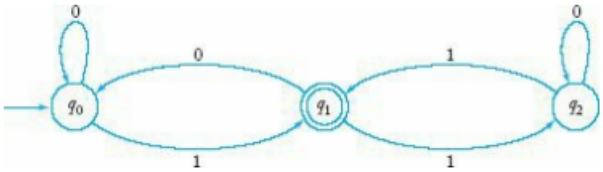

Figure 2.1: Example of a basic DFA.

This DFA accepts the string `01`. Starting in state q_0, the first symbol, `0`, is read, and the automaton remains in state q_0. Upon reading the next symbol, `1`, it transitions to state q_1. Since the input string has been fully consumed and the automaton is now in the final state q_1, the string is accepted. Conversely, the string `00` is not accepted. After processing two consecutive `0`'s, the machine remains in state q_0, which is not a final state. By similar reasoning, the automaton accepts strings such as `101`, `0111`, and `11001`, but not `100` or `1100`.

2.3.2 Nondeterministic Finite Automata (NFAs)

A Nondeterministic Finite Automaton (NFA) is the general case of finite automata, where there are not restrictions on the amount of transitions with the same input symbol from the same state, as well as λ-transitions. It is defined as a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F),$$

where:

- Q is a finite set of states,
- Σ is the input alphabet,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of final states,
- δ is the transition relation, a finite subset of

$$(Q \times (\Sigma \cup \{\epsilon\})) \times Q.$$

Essentially, each element of δ contains a pair consisting of a state and an input symbol (or ϵ), along with the corresponding next state. In a nondeterministic automaton, the value of the transition function is not a single element of Q but a subset of it. It defines the set of all possible states that can be reached by the transition.

For example, if the current state is q_1, the input symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA.

λ (or ϵ) are accepted as the second argument of δ, meaning the NFA can make transitions without consuming an input symbol. It is also possible for it to remain in the same state during a λ-transition. Also, in an NFA, the set $\delta(q_i, a)$ can be empty, indicating that there is no transition defined for that specific combination of state and input symbol. A string is accepted by an NFA if there exists some sequence of possible moves that places the machine in a final state at the end of the string. However, a string is rejected if there is no sequence of moves that can reach a final state. So we can look at nondeterminism as a mechanism by which the machine can always choose the correct path to acceptance, if one exists. Consider the following example of the above described transition graph in Figure 1.1. It describes a nondeterministic automaton since there are two transitions labeled a out of q_0.

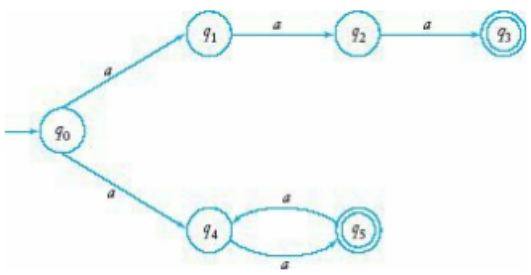

Figure 2.2: Example of a basic NFA.

2.4 Finite-State Transducers

Finite-State Transducers (FSTs) extend the concept of finite automata by associating output symbols with transitions, enabling the transformation of input strings into output strings. They are primarily classified as recognizers even though they generate an output. An FST is associated with an input that contains a string from an input alphabet Σ, , and an output mechanism that produces a string from an output alphabet Γ in response to a given input In FSTs, the concept of a configuration is also used, similar to finite automata, however, in this case, a configuration is defined as a 3-tuple (q,w,o), where q is the current state, w is the remaining input word, and o is the output produced so far.

2.4.1 Operations on Finite-State Transducers

The following operations, which are also defined for finite automata, apply to finite-state transducers:[17]

- **Union** Given transducers T and S, there exists a transducer $T \cup S$ such that:

$$x[T \cup S]y \iff x[T]y \quad \text{or} \quad x[S]y.$$

- **Intersection** Given transducers T and S, there exists a transducer $T \cap S$ such that:

$$x[T \cup S]y \iff x[T]y \quad \text{and} \quad x[S]y.$$

- **Concatenation** Given transducers T and S, there exists a transducer $T \cdot S$ such that:

$$x[T \cdot S]y \iff \exists x_1, x_2, y_1, y_2 \text{ with } x = x_1x_2,\ y = y_1y_2,\ x_1[T]y_1 \text{ and } x_2[S]y_2.$$

- **Kleene Star** Given a transducer T, there might exist a transducer T^* such that:

  (k1) $\varepsilon[T^*]\varepsilon$;

  (k2) if $w[T^*]y$ and $x[T]z$, then $wx[T^*]yz$;

  (k3) and $x[T^*]y$ holds only if implied by (k1) or (k2).

The following operations are specific to finite-state transducers:

- **Composition** Given a transducer T on alphabets Σ and Γ and a transducer S on alphabets Γ and Δ, there exists a transducer $T \circ S$ on Σ and Δ such that:

$$x[T \circ S]z \iff \exists y \in \Gamma^* \text{ such that } x[T]y \text{ and } y[S]z.$$

- **Projection to an Automaton** There are two projection functions:

  – π_1 preserves the input tape. Given a transducer T, there exists a finite automaton $\pi_1 T$ such that:

$$\pi_1 T \text{ accepts } x \iff \exists y \text{ such that } x[T]y.$$

  - π_2 preserves the output tape, and is defined analogously.

- **Determinization** Given a transducer T, the goal is to construct an equivalent deterministic transducer—one with a unique initial state and no two transitions from any state sharing the same input label. The determination algorithm used can be extended to the transducer, though in some cases it may not terminate. Not all non-deterministic transducers admit equivalent deterministic versions. Characterizations of determinizable transducers have been proposed, along with algorithms for testing determinization.

- **Minimization** There are several algorithms for minimizing transducers.

The two most commonly studied cases of FSTs are Mealy machines and Moore machines, which differ in how they produce the output based on the current state and input symbol. Later we will also see more about the general FSTs.

2.4.2 Mealy Machines

In a Mealy machine, the output produced by each transition depends on the current internal state and the input symbol used in the transition, so the output is generated during the transition itself.

A Mealy machine is defined as a 6-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, \theta, q_0),$$

where:

- Q is a finite set of internal states,
- Σ is the input alphabet,
- Γ is the output alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function, which is a total function,
- $\theta : Q \times \Sigma \to \Gamma$ is the output function,
- $q_0 \in Q$ is the initial state of M.

The machine starts in the initial state q_0, with the entire input available for processing. If at time t_n, the Mealy machine is in state q_i, the current input symbol is $a \in \Sigma$, and

$$\delta(q_i, a) = q_j, \quad \theta(q_i, a) = b,$$

then the machine will transition to state q_j and produce output symbol $b \in \Gamma$. The process continues until the end of the input is reached.

Note that there are no final or accepting states associated with a transducer such as the Mealy machine, since the focus is on input-output transformations rather than language acceptance. The finite-state transducer (FST) is defined by the following components:

$$Q = \{q_0, q_1\}, \quad \Sigma = \{0, 1\}, \quad \Gamma = \{a, b, c\},$$

with initial state q_0 and

$$\begin{aligned} \delta(q_0, 0) &= q_1, \quad \delta(q_0, 1) = q_0, \\ \delta(q_1, 0) &= q_0, \quad \delta(q_1, 1) = q_1, \\ \theta(q_0, 0) &= a, \quad \theta(q_0, 1) = c, \\ \theta(q_1, 0) &= b, \quad \theta(q_1, 1) = a. \end{aligned}$$

This FST is represented by the graph shown in Figure 1.3. This Mealy machine prints the output string `caab` when given the input string `1010`.

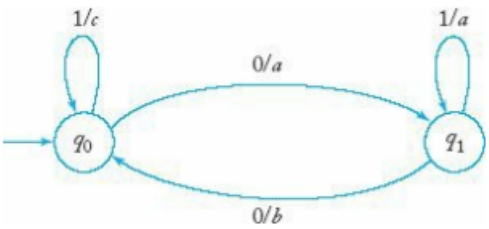

Figure 2.3: Example of a basic mealy machine.

2.4.3 Moore Machines

Moore machines differ from Mealy machines in the way output is produced. In a Moore machine, every state is associated with an element of the output alphabet. When the machine enters a given state, the output symbol is then produced. The output is generated only when a state transition occurs, so the symbol associated with the initial state is not printed at the start, but may be produced if the initial state is re-entered later.

A Moore machine is defined as a 6-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, \theta, q_0),$$

where:

- Q is a finite set of internal states,
- Σ is the input alphabet,
- Γ is the output alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function, which is a total function,

- $\theta : Q \rightarrow \Gamma$ is the output function,
- $q_0 \in Q$ is the initial state.

In the transition graph of a Moore machine, each state has two labels: its name and the associated output symbol.

Here we have a Moore machine M that takes strings of 0's and 1's as input. The output is a string of 0's until the first 1 occurs in the input, at which time it will switch to printing 1's. This continues until the next 1 occurs in the input, when the output reverts to 0. The alternation continues every time a 1 is encountered. For example, FM (0010010) = 0011100. This FST is a simple model for a flip-flop circuit.

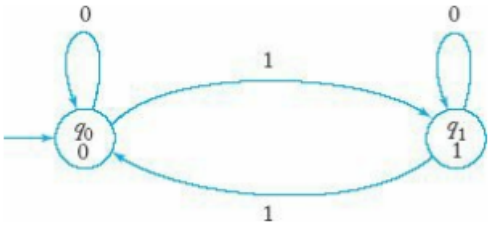

Figure 2.4: Example of a basic Moore machine.

2.4.4 Moore and Mealy Machine Equivalence

The examples in the previous sections showcase the differences between Moore and Mealy machines, but more importantly it allows us to see that they can be used to implement the same functions. In this sense, the two types of transducers can be considered equivalent.

Two finite-state transducers M and N are said to be equivalent if they implement the same function.[5]

The conversion of a Moore machine to an equivalent Mealy machine is straightforward. The states of the two machines are the same, and the symbol that is to be printed by the Moore machine is assigned to the Mealy machine transition that leads to that state.

The conversion from a Mealy machine to an equivalent Moore machine is more involved, because the states of the Moore machine must carry two pieces of information: the internal state of the corresponding Mealy machine and the output symbol produced by the Mealy machine's transition to that state.

In the construction, we create, for each state q_i of the Mealy machine, $|\Gamma|$ states in the Moore machine, labeled q_i^a for each $a \in \Gamma$. The output function for the Moore machine states is defined as

$$\theta(q_j^a) = a.$$

When the Mealy machine transitions to state q_j and produces the output symbol a, the equivalent Moore machine transitions into state q_j^a, thereby producing the same output a.

The Mealy machine in Figure 1.5(a) and the Moore machine in Figure 1.5(b) are equivalent.

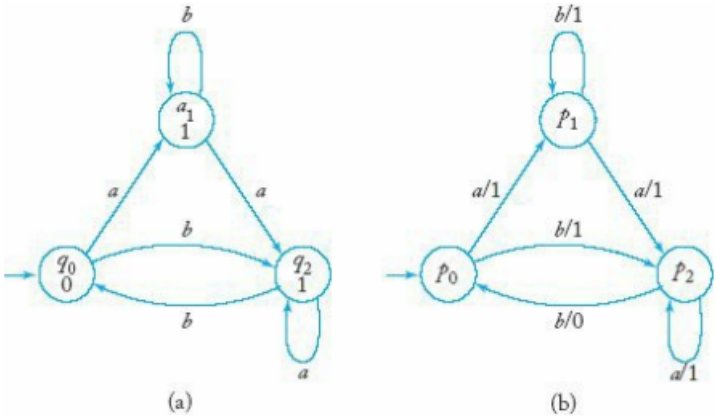

Figure 2.5: Example of a Mealy machine and its equivalent Moore machine.

2.4.5 Mealy Machine Minimization

For a given function $\Sigma^* \rightarrow \Gamma^*$, there are often many equivalent finite-state transducers, some of them differing in the number of internal states. For practical purposes, it is often important to find the minimal FST, that is, the machine with the smallest number of internal states.

The first step in minimizing a Mealy machine is to remove states that play no role in any computation because they cannot be reached from the initial state. When there are no inaccessible states, the FST is said to be connected. However, a Mealy machine can be connected and still not minimal, as the following example illustrates.

The Mealy machine shown in Figure 1.6(a) is connected, but it is clear that the states q_1 and q_2 serve the same purpose and can be combined to produce the machine in Figure 1.6(b).

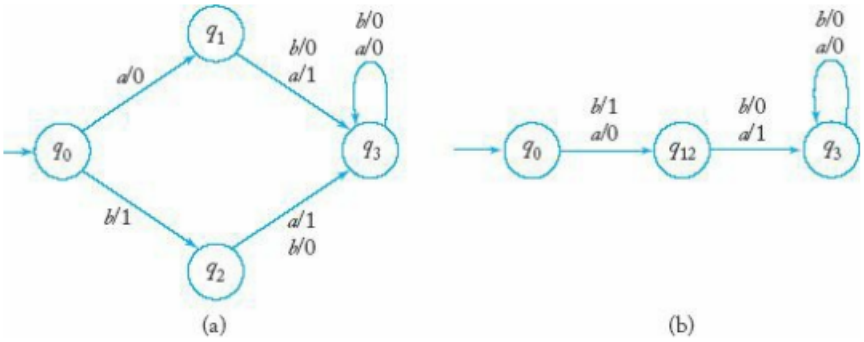

Figure 2.6: Example of a mealy machine and its minimal equivalent.

The minimization of a Mealy machine begins by identifying equivalent states—states that cannot be distinguished by any input string.[5]

Partition Algorithm

1. **Initial 1-equivalence partition:** Given the set of states $Q = \{q_0, q_1, \ldots, q_n\}$, determine all states that are 1-equivalent to q_0 and partition Q into sets:

   $$\{q_0, q_i, \ldots, q_j\} \quad \text{and} \quad \{q_k, \ldots, q_l\},$$

   where the first set contains all states 1-equivalent to q_0 and the second those 1-distinguishable from it. Repeat this process with each state in Q, merging duplicates to form an initial partition based on 1-equivalence.

2. **Refinement:** For every pair of states q_i, q_j within the same equivalence class, examine their transitions. If there exist transitions on an input symbol leading to states in different equivalence classes, the current class to separate q_i and q_j. Check all pairs accordingly.

3. **Repeat:** Continue refining partitions until no further splits occur.

After the algorithm terminates, Q will be partitioned into equivalence classes $E_1, E_2, \ldots, E_m$, with all states within each class being indistinguishable.

Constructing the Minimal Mealy Machine

A Mealy machine $M = (Q, \Sigma, \Gamma, \delta, \theta, q_0)$, we construct its minimal equivalent machine $P = (Q_P, \Sigma, \Gamma, \delta_P, \theta_P, E_p)$ as follows:

- Define the state set of P as the set of equivalence classes found in the partition: $Q_P = \{E_1, E_2, \ldots, E_m\}$.
- For each class E_r and input symbol $a \in \Sigma$, select any state $q_i \in E_r$. If $\delta(q_i, a) = q_j$ and $\theta(q_i, a) = b$, define the transitions in P by

  $$\delta_P(E_r, a) = E_s \quad \text{where } q_j \in E_s,$$

  and

  $$\theta_P(E_r, a) = b.$$

- The start state of P is the equivalence class E_p containing the original start state q_0.

The resulting machine P is minimal and equivalent to M.

2.4.6 Moore Machine Minimization

The minimization of a Moore machine largely follows the same procedure used for Mealy machines, but with some differences due to the nature of Moore machines. In a Moore machine, outputs are associated with states rather than transitions. Therefore, states with different outputs are immediately 0-distinguishable (distinguishable without reading any input), and partitioning must start by separating states with different output symbols.[5] Moore machines: two states q_i and q_j are

- **0-distinguishable** if their output symbols differ;
- **k-distinguishable** if they were already $(k-1)$-distinguishable or if, for some input $a \in \Sigma$, their transitions lead to $(k-1)$-distinguishable states.

The partitioning algorithm proceeds as with Mealy machines: repeatedly refining equivalence classes until no further distinctions are possible. Once stable equivalence classes are found, the minimal Moore machine P is constructed by:

- defining states of P as the equivalence classes $E_1, E_2, \ldots, E_m$,
- setting transitions so that if $\delta(q_i, a) = q_j$ with $q_i \in E_r$ and $q_j \in E_s$, then

  $$\delta_P(E_r, a) = E_s,$$

  and the output function by

  $$\theta_P(E_s) = \theta(q_j),$$

  using any representative q_j from the class E_s.

A complication arises in Moore machines however. The initial state's output may not be uniquely determined when the initial state cannot be re-entered, potentially resulting in minimal machines that are not identical under simple state relabeling. However, this issue is minor and does not affect correctness.

2.4.7 General Finite-State Transducers

The general case for finite-state transducers includes non-determinism, this introduces important differences compared to the two deterministic transducer types previously discussed, which were very special cases of finite state transducers where the notion of word acceptance or rejection are not used, instead focusing on generating output from an input. In non-deterministic FSTs, a single input symbol may lead to multiple possible transitions from a given state, each potentially producing different outputs or no output at all. Furthermore, operations such as determinization, minimization, and output tracing become more involved or even undecidable in certain cases. Formally, a finite-state transducer T is a 6-tuple: [17]

$$T = (Q, \Sigma, \Gamma, I, F, \delta)$$

where:

- Q is a finite set, the set of **states**;
- Σ is a finite set, called the **input alphabet**;
- Γ is a finite set, called the **output alphabet**;
- $I \subseteq Q$ is the set of **initial states**;

- $F \subseteq Q$ is the set of **final states**;

- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q$ is the **transition relation**, where ε represents the empty string.

As we have a transition relation instead of a transition function, multiple transitions for the same input symbol from a single state are allowed, as well as λ symbols both in the input and output.

Determination of Non-deterministic FSTs

The determinization of finite-state transducers (FSTs) is an extension of the determinization algorithm used for finite automata. Where each state in the resulting deterministic transducer corresponds to a set of original states, each paired with the symbols that are yet to be emitted. This added complexity is necessary because, unlike automata, transducers must handle both input recognition and output generation.

The result of the determinization process is typically a form of deterministic transducer that permits output strings to be associated with final states. When computation halts in such a state, the associated output is appended to the result. This design allows the transducer to emit outputs conditionally, depending on whether it has reached the end of the input string.

However, not all FSTs are determinizable, even if they define a function. In some cases, determinization fails because the decision on which path to take requires an unbounded lookahead. For example, in a given transducer, the system might need to read an unlimited number of symbols before choosing the correct transition. This leads to infinite branching in the determinization process, effectively causing the algorithm to loop endlessly without producing a result. Figure 2.7 is an example of a non-deterministic FST that cannot be determinized: [10]

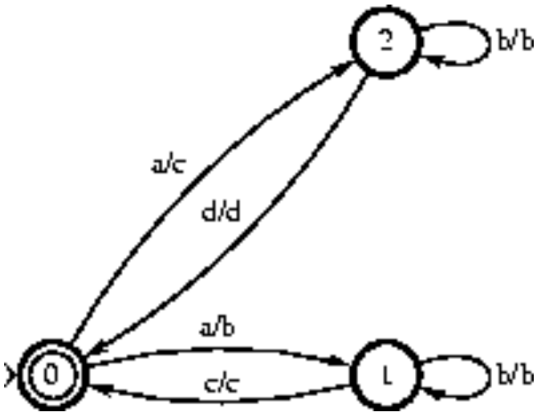

Figure 2.7: Example of a non-deterministic FST that cannot be determinized.

If the system is in state 0 and it reads an a, then it can only decide which transition to take after reading an unbounded number of b's. The algorithm enters an infinite loop for transducers for which no determinized version exists.

2.5 Conclusion

Finite-State Transducers derive from foundational models of computation and extend the capabilities of finite automata by introducing output transformations. Their theoretical basis is related to regular languages (Type 3 of Chomsky's hierarchy). We also explored the two main types of FSTs, Mealy and Moore machines, which differ in how they produce output. Also explored their equivalence, minimization techniques. Also touched on the challenges of determinization and minimization of non-deterministic FSTs by looking at a general definition of FST. By understanding these concepts, it is now possible to delve into the specifics of FSTs and their applications, such as in the OFLAT application.

3

Related Work

3.1 Introduction

In this chapter, we will explore existing pedagogical tools for teaching Formal Languages and Automata Theory (FLAT), including ones that provide support for finite-state transducers (FSTs). As well as comparing them with each other, and with the OFLAT tool. This is done in order to contextualize the work to be done.

3.2 JFLAP: Java Formal Languages and Automata Package

JFLAP (Java Formal Languages and Automata Package)[14] is a widely adopted educational tool for teaching and learning formal languages and automata theory. Initially developed in the 1990s by Professor Susan H. Rodger and her students at Rensselaer Polytechnic Institute, JFLAP was first created in C++ and X Window under the name FLAP. It was later rewritten in Java and renamed JFLAP, enhancing its portability and usability. As a Java application, it runs on most systems equipped with the Java Runtime Environment (JRE).

JFLAP offers a comprehensive suite of features supporting the construction, simulation, testing, and conversion of theoretical models. JFLAP supports finite automata, formal grammar manipulation, including regular, context-free, unrestricted grammars, turing machines, etc. And can convert between multiple of these models.

The tool provides immediate visual feedback by allowing users to simulate automata step by step, highlighting state transitions and input consumption. Figure 3.1 illustrates the model option in JFLAP.

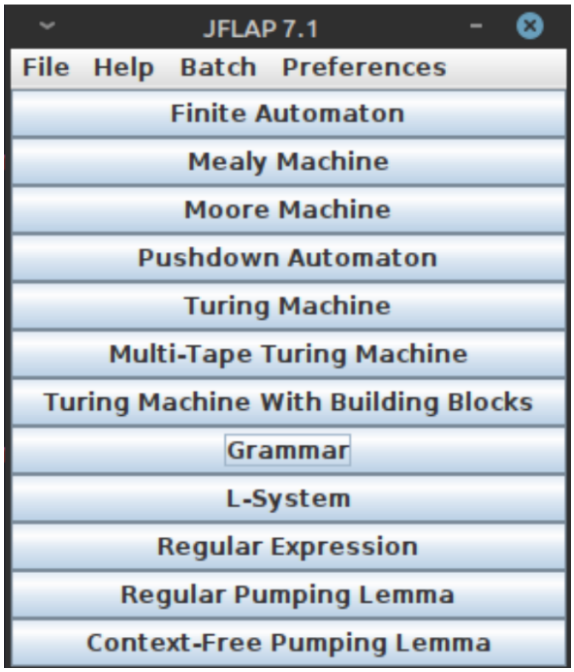

Figure 3.1: Model list available in JFLAP.

JFLAP also offers support for finite-state transducers in the form of Mealy and Moore machines, it has simulation and visualization, allowing users to observe state transitions alongside produced outputs. Figure 3.2 shows a simple Mealy machine in JFLAP,.

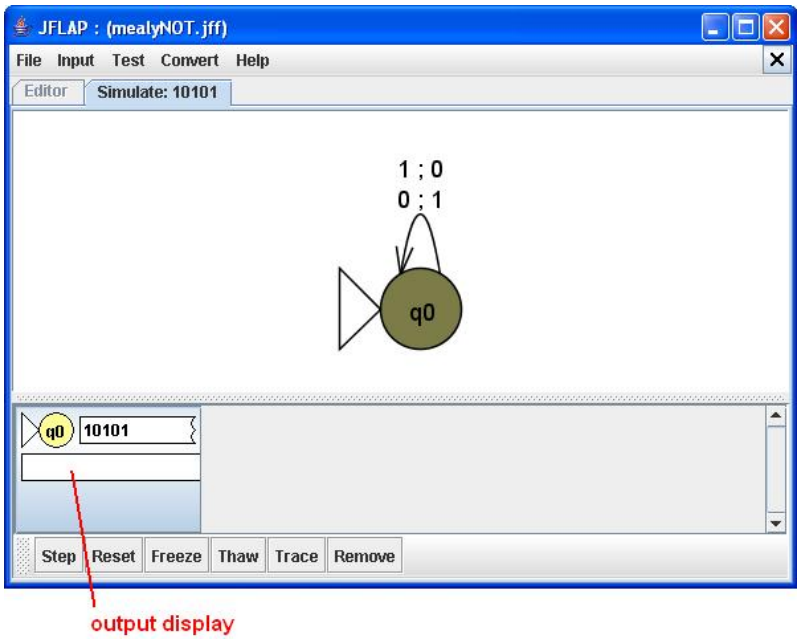

Figure 3.2: A simple Mealy machine in JFLAP.

This example shows us a Mealy machine includes a single state, `q0`, which serves as both the initial and active state throughout the computation. The transitions are represented graphically by an arc, and are labeled in the form `input ; output`. In the

example, we can see the transition with an input of `1` produces an output of `0`, and vice versa. The input string `10101` provided below the state diagram, and the machine's output is displayed dynamically as the simulation progresses. The interface includes a control panel with options such as `Step`, `Reset`, `Trace`, and others, allowing users to simulate the machine step-by-step.

Conclusion

JFLAP distinguishes itself as one of the most complete and pedagogically effective tools in formal language education. It supports a wide variety of computational models and conversions grounded in formal theory.

3.3 FAdo

Similar to OFLAT, FAdo[13] is a Portuguese project that aims to develop an interactiv environment for the symbolic manipulation of formal languages. It consists of a Python library designed for symbolic manipulation of finite automata and regular expressions. It supports finite automata, providing minimization, equivalence checking, language operations (union, intersection, complementation), and conversion between automata and regular expressions.

In contrast to interactive graphical tools such as JFLAP, OFLAT, and Automaton Simulator, the notion of acceptance in FAdo is more limited in scope, particularly with regard to visualization and user interaction. FAdo is primarily intended to be used programmatically within a Python interpreter or in script-based workflows. As such, it does not provide a graphical user interface or native support for animated, step-by-step execution of automata. Instead, the FAdo library emphasizes flexibility and extensibility within a code-based environment. To determine whether a word is accepted by a defined finite automaton, users can invoke the method evalWord(word), which simply returns a Boolean value (True or False). This binary output informs whether the automaton, in its current configuration, accepts the given input string. While this functionality is effective for bulk testing or automated workflows, it does not provide insight into how the automaton processes the string or transitions through its states during evaluation. For users who wish to simulate the acceptance process manually, FAdo provides a lower-level function, evalSymbol(init, symb). This function allows for inspection of individual transitions by returning the resulting state reached when consuming a symbol symb from a current state init. By iteratively applying evalSymbol for each symbol in the input string—updating the initial state to be the result of the previous call—a user can effectively simulate the path that the automaton takes through its states.

FAdo also provides support for finite-state transducers however, it is not publicaly available at the moment.[3]

However, this manual step-by-step execution is not integrated into the library as a built-in feature. In educational contexts, this limitation makes FAdo less suited for beginners or for visual demonstrations of automaton behavior. However, its strength lies in its integration into Python's ecosystem, which enables programmatic construction, manipulation, and analysis of automata and formal languages.

3.4 Automaton Simulator

The Automaton Simulator[4] is an open-source, browser-based tool designed for constructing and simulating various types of automata. It supports deterministic finite automata (DFA), nondeterministic finite automata (NFA), and pushdown automata (PDA), which offers a simple and interactive environment for educational purposes.

The interface is divided into several functional sections that enhance usability. On the left-hand side Figure 3.3, users can input strings for testing. There are two main boxes: one labeled `Accept (one per line)` for input strings expected to be accepted by the automaton, and one labeled `Reject (one per line)` for strings expected to be rejected. These strings can be tested in bulk using the *Bulk Testing* feature, which evaluates all entries at once and displays their acceptance or rejection. There is also a `Test/Debug` field, which allows for dynamic simulation.

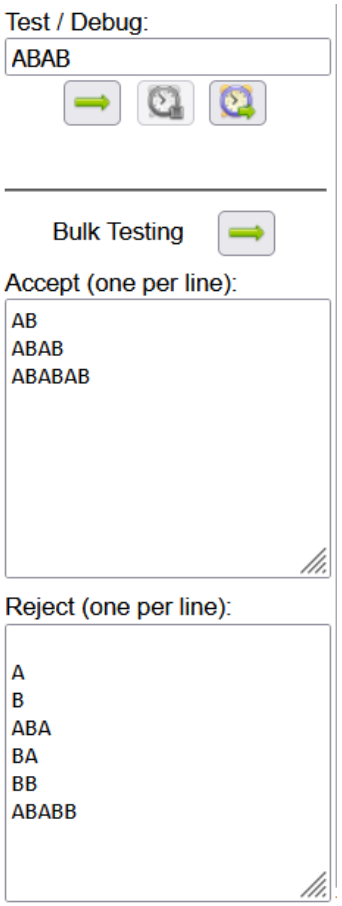

Figure 3.3: Left panel of the Automaton Simulator.

Below these input areas is the `Test Results` panel, where detailed feedback is shown after the test runs. Above the input section, buttons for loading, saving, and resetting the automaton are displayed. The central canvas of the interface is where the automaton is constructed Figure 3.4. States are created by clicking, and transitions are added by clicking and dragging between states. The simulation highlights the current state using a different color, the transition being taken, and the unprocessed portion of the input string. This is especially useful in step-by-step simulations where the learner can observe the exact sequence of transitions being executed.

Each state is displayed as a rounded rectangle, with different markings to indicate its role:

- A checked box denotes an accepting (final) state.
- Transitions are labeled with the input symbol that triggers them.

The simulator provides feedback on acceptance using textual cues such as "Accepted" or "Rejected", displayed near the top-left corner of the interface. During step-by-step execution, this feedback updates in real time as the input string is consumed.

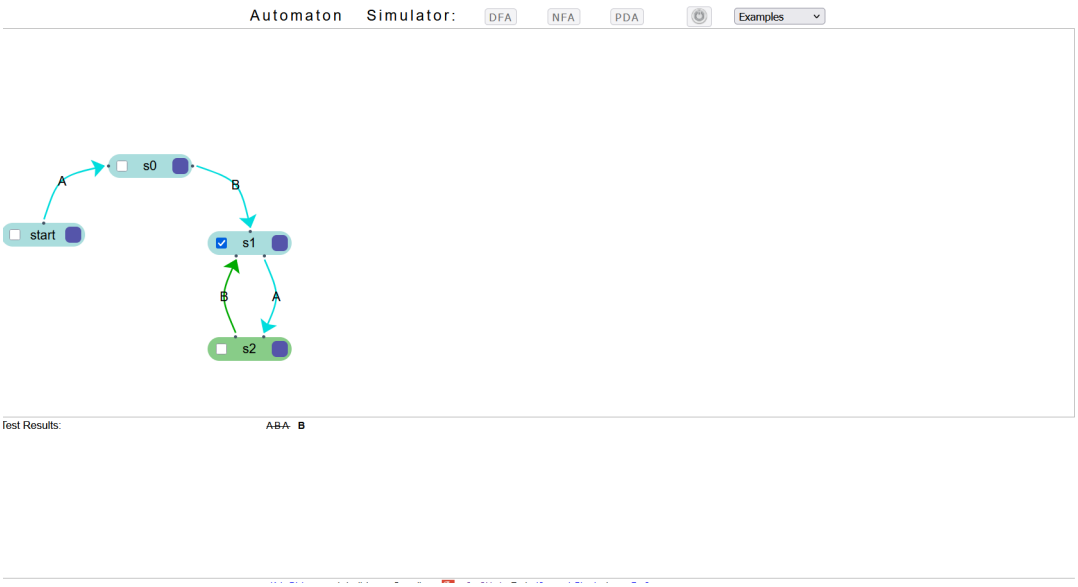

Figure 3.4: Central area of the Automaton Simulator.

Conclusion

The Automaton Simulator provides an effective and user-friendly platform for constructing and simulating automata directly in the web browser. Its strength lies in its simplicity and immediate feedback mechanism, which makes it a good tool for introductory coursework. Although it lacks the extensive model coverage and transformation tools found in more comprehensive platforms like JFLAP or OFLAT.

3.5 Flap.js

Flap.JS[2] is a web application therefore easily available on all kinds of devices, including mobile devices. It supports the graphical creation of Finite Automata, Regular Expressions, and Pushdown Automata, as well as tests and conversions, but none of these use animations, which would improve the learning process. In their web application, we can see that support for finite state transducers is not available, however, Mealy machines, Moore machines and turing machines are greyed out, indicating that they are planned features to be added in the future. Figure 3.5 shows the user interface, which while unusual can be a way to get perspective on different design approaches when it comes to graphical interfaces.

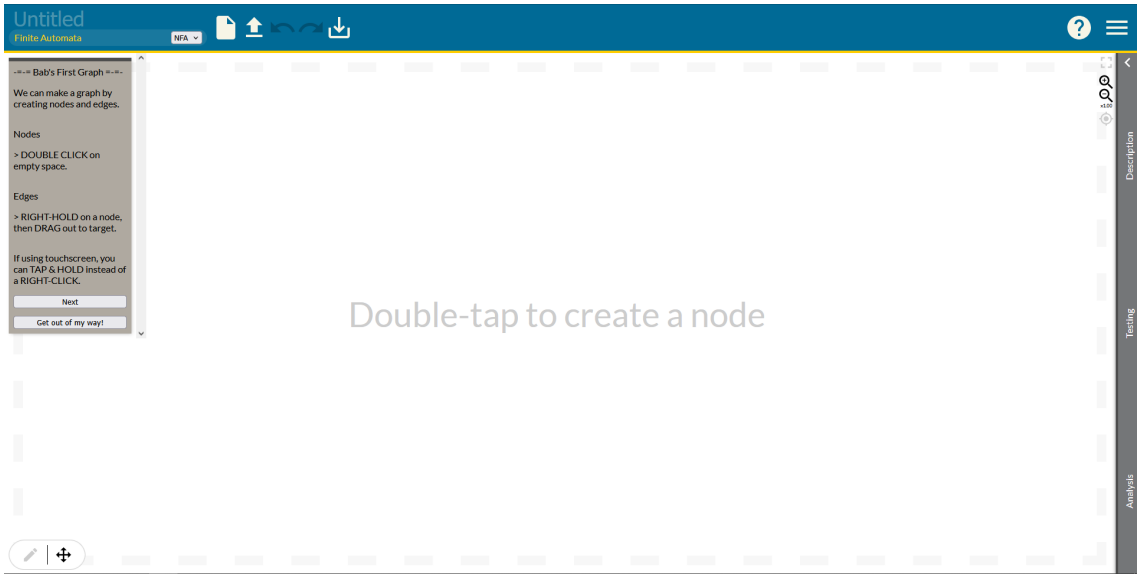

Figure 3.5: Flap.js user interface.

Conclusion

Flap.js, as it stands, does not provide as much variety in its current state as some of the others, like JFLAP and our OFLAT. As well as having a less intuitive interface, then Automaton Simulator, which is simpler and provides similar functionalities.

4

OCamlFLAT / OFLAT

This chapter presents the OCamlFLAT/OFLAT system as it stands currently. The system is built of two parts, the web application, which offers a graphical interface to the system, and the library it is built upon, which contains the logical part of the system. By understanding the OCamlFLAT/OFLAT system's design and limitations, we can then have a better understanding of the new functionalities to be added.

4.1 OCamlFLAT Library

The OCamlFLAT library is structured to provide a clear and modular framework for working with formal languages and automata. Its hierarchical module design starts with abstract concepts at the top, simplifying code organization and future expansions. At the top is the **Entity** module, which generalizes all other modules. The **Exercise** module supports exercise creation and validation, while the **Model** module offers shared utilities for all FLAT models, encouraging consistent implementations and code reuse. In this thesis, support for finite state transducers will be added to the library as well as all its necessary operations, which will be implemented as a new module.

Supporting modules handle essential tasks such as JSON serialization, error management, and foundational type definitions. Among these, the **BasicTypes** module plays a key role by defining core constructs like symbols, words, states, and transitions—elements central to modeling automata and formal languages.

Each formal model in the library is represented by a pair of modules: one for auxiliary tasks like parsing and formatting, and another dedicated to core logic. The main logic modules include:

- **Regular Expression Module**: Implements word generation, word acceptance checking, conversion of regular expressions into finite automata, and simplification routines.
- **Finite Automaton Module**: Provides functions for creating, editing, minimizing, and analyzing finite automata, including conversions between deterministic and nondeterministic forms and detection of useful states.

- **Context Free Grammar Module**: Offers support for word generation and acceptance, validation, removal of left recursion and epsilon productions, and transformations to LL(1) grammars. It also provides computation of First and Follow sets, parsing table construction, and support for LR(0), SLR(1), LR(1), and LALR(1) parsing techniques.
- **Pushdown Automaton Module**: Extends FA functionalities to pushdown automata, covering acceptance, generation, and minimization, with mechanisms for validating determinism and identifying productive and reachable states.
- **Turing Machine Module**: Handles creation, modification, and analysis of Turing machines, including validation, determinism checking, linear boundedness checking, and conversions to finite automata where appropriate.
- **Composition Module**: Supports operations on model compositions, such as union, concatenation, intersection, and Kleene closure.
- **Poly Module**: Facilitates type conversions and interactions between different model types, including conversions across regular expressions, finite automata, context-free grammars, PDAs, and TMs.
- **Exercises Module**: Manages definition and manipulation of exercises, aligning with the system's pedagogical goals.

4.2 OFLAT

OFLAT is a web-based application providing an interactive interface built upon the OCamlFLAT library, with an emphasis on ease of use and educational effectiveness. Its design allows users to intuitively create, manipulate, and analyze a wide variety of formal language models.

4.2.1 User Interface Layout

OFLAT's interface is split into two primary sections:

1. **Side Menu**: Located on the left (shown in Figure 4.1), this menu lets users choose models such as finite automata, regular expressions, context-free grammars, pushdown automata, and Turing machines. It also offers predefined examples for quick loading, options for word operations (acceptance, generation, tracing, conversions), and the ability to import or export models.
2. **Dynamic Workspace**: On the right side (shown in Figure 4.2), this area dynamically updates based on user interactions. It displays the current model visually and provides tools for editing, such as adding or removing states, transitions, or modifying grammar rules. The workspace often splits into a left panel for model structure and operations, and a right panel for operation results.

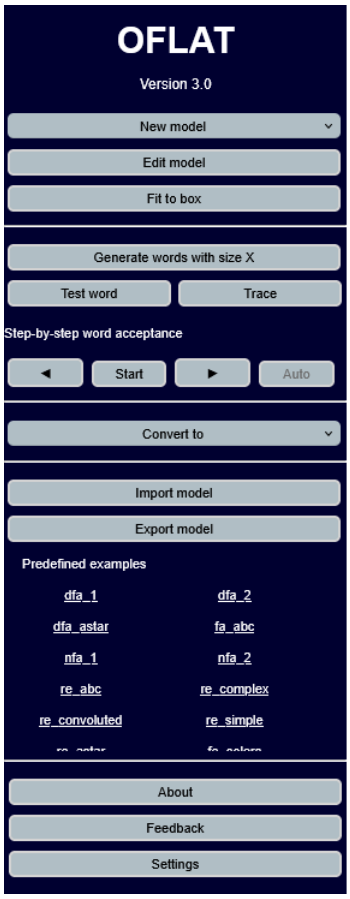

Figure 4.1: OFLAT Side Menu

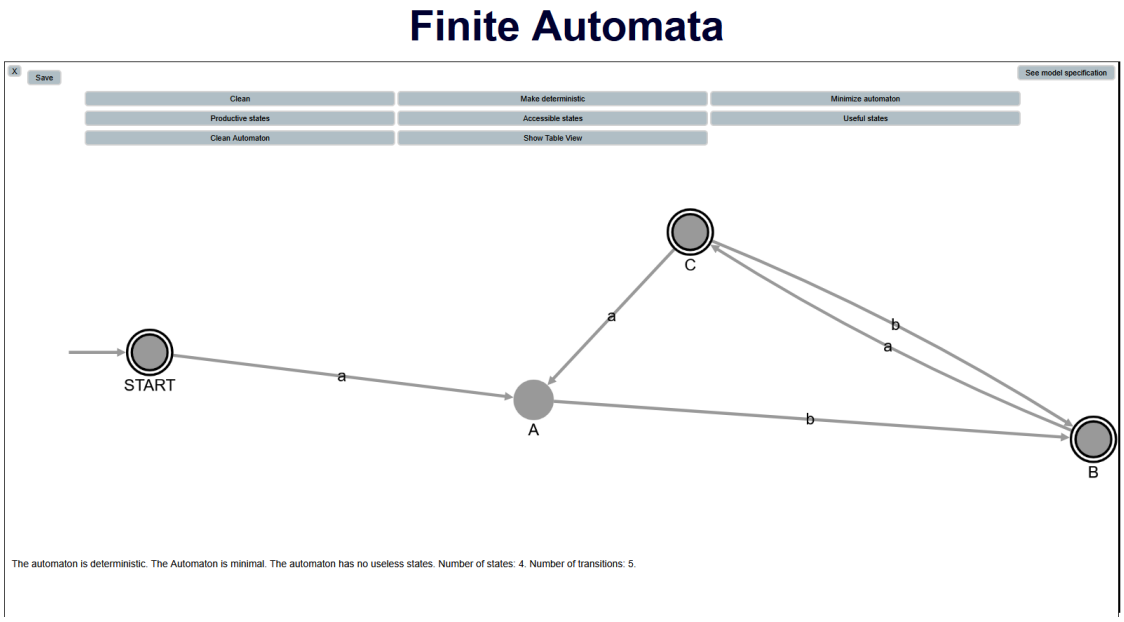

Figure 4.2: OFLAT Workspace showing a finite automaton

4.2.2 OFLAT Architecture

The OFLAT web application follows the Model-View-Controller (MVC) design pattern, which ensures a clear separation of responsibilities among its components:

- **Model**: The `OCamlFLAT` library acts as the model layer, handling all the logic and data structures related to formal language theory.
- **View**: This layer includes the graphical components of the system. The view modules are responsible for the visual representation of models and for handling user interaction.
- **Controller**: The controller modules process user input and turn them into commands for the view and model.

A standard operation in OFLAT typically follows this sequence:

1. The user interacts with the interface by providing input.
2. The controller receives the user command and, using the model (i.e., the OCamlFLAT library), performs the required computations.
3. The controller then updates the view to display the results.

This modular design enhances the system's maintainability and makes it easier to extend with new features or support for additional formal models in the future.

4.3 Conclusion

Together, the OCamlFLAT library and the OFLAT web interface, provide a platform for learning and exploring formal language concepts through interactive, evolving tools. While the system has already achieved significant functionality, further enhancements are necessary to meet its full potential. For instance, the absence of a help section and the sometimes overwhelming arrangement of buttons in the interface can hinder user experience. Nevertheless, OFLAT supports future growth, allowing new models and functionalities to be added.

5

Technologies

This chapter provides an overview of the key technologies used in the system. We present the OCaml programming language, the mechanisms for interoperability between OCaml and JavaScript, and the use of additional libraries (such as Cytoscape.js) that enable both the functional and graphical aspects of the platform.

5.1 OCaml Programming Language

OCaml is a general-purpose, high-level, multi-paradigm programming language that extends the Caml dialect of ML with object-oriented features. It was created in 1996 by Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy, Ascánder Suárez, and others at the French Institute for Research in Computer Science and Automation (INRIA). The functional programming fragment of OCaml is based on the concept of expressions, where computations are performed by evaluating expressions that produce values, naturally supporting a declarative programming paradigm. [11]

5.1.1 Declarative Programming

Declarative programming focuses on writing code that solves problems by describing their logical properties rather than by elaborating an algorithm filled with operational details. In declarative programming[16], the emphasis is on specifying what the program should accomplish, rather than how to achieve it step by step.

For example, to determine the length of a list, a declarative solution might consist of a simple statement of the property: "the length of a list is one plus the length of its tail." In contrast, an imperative solution would typically involve initializing a counter to zero and then traversing the list, incrementing the counter for each element. Declarative programming is often considered very high-level because it is closer to mathematical methods and human thought processes, as opposed to programming in a way that mimics machine operations.

Recursive functions written in a declarative style are always inductive functions, in which the problem is reduced to smaller instances of the same problem. This makes many

complex problems easier to solve. The code also becomes more readable, as it resembles an equation and does not require mentally simulating its execution to understand it. When writing inductive functions, it greatly helps if the data themselves are inductive in nature—for example, using lists or trees. There are situations where writing a declarative solution still requires sophisticated considerations. For instance, to search for a value in an infinite binary tree, one must integrate the need to perform a breadth-first traversal into the solution. The exploration of infinite data structures is frequent in this project, as the configuration spaces that need to be explored are often infinite.

5.2 Interoperability using js_of_ocaml

One of the fundamental technologies enabling the OCamlFLAT/OFLAT system to function as a web-based tool is the `js_of_ocaml` framework. js_of_ocaml[12] is the interoperability solution used for bridging OCaml and JavaScript. Enabling the use of OCaml's type system and functional programming features in web development. With this we can code the entire tool in OCaml. js_of_ocaml is not only a compiler but a collection of tools that includes a library, a binding mechanism, and a syntactic extension.

Compiler

At the core of `js_of_ocaml` is its compiler, which translates OCaml code into JavaScript code. This compilation step typically follows the standard OCaml compilation pipeline: the source code is first compiled using `ocamlc`, and the result is then passed to `js_of_ocaml` for translation into JavaScript. This allows us to have the type safety and early error detection provided by OCaml.

5.2.1 Bindings

Bindings is a crucial component for enabling interoperability between OCaml and JavaScript. In the `js_of_ocaml` ecosystem, bindings are used extensively to connect both languages, allowing OCaml code to access and manipulate JavaScript variables, functions and prototypes.

For example, the JavaScript alert function can be made available from OCaml using the following binding:

```
let alert (msg: string): unit = Js.Unsafe.global##alert (Js.string msg)
```

This snippet wraps the browser's window.alert in an OCaml function, converting the OCaml string to a JavaScript string before invocation.

As a second example, the following definition makes available on the OCaml side, the JavaScrip constructor for the Cytoscape objects (that represents graphs).

```
let cytoscape_constructor: Js.Unsafe.t = Js.Unsafe.pure_js_expr "Cytoscape"
```

The following OCaml type, as shown in Listing 5.1, describes the structure of the Cytoscape objects, and it allows method invocation with type safety:

Listing 5.1: Binding for Cytoscape object

```
class type cytoscape =
object
  method add : DataItem.t t -> unit meth
  method remove : DataItem.t t -> unit meth
  method remove_fromSelector : js_string t -> unit meth
  method mount : Dom_html.element t -> unit meth
  method layout : layout_options t -> layout t meth
  method resize : unit meth
  (*...*)
  method edges : js_string t -> DataItem.t Js.t js_array Js.t meth
  method nodes : js_string t -> DataItem.t Js.t js_array Js.t meth
  method tapdragover : DataItem.t t -> unit meth
  method contextMenus : 'c t -> contextMenus Js.t meth
  method popper : 'z -> popper Js.t meth
end
```

Syntax Extensions

To make JavaScript interoperation more idiomatic in OCaml, `js_of_ocaml` provides syntax extensions that support type-safe method invocation and property access. These extensions enable a natural coding style that mimics JavaScript syntax while remaining within OCaml's type system.

Library

Beyond the compiler and bindings, `js_of_ocaml` includes a library that exposes a substantial portion of the browser's API. This includes modules for:

- DOM manipulation (`Dom_html`, `Dom`)
- Event handling
- HTML5 APIs
- Canvas and graphics operations

5.3 Cytoscape.js for Graph Visualization

Cytoscape.js is an open-source JavaScript library for graph theory analysis and visualization [1]. In the context of the OFLAT system, it is used to render automata and tree structures graphically, providing users with an intuitive and interactive interface to construct and manipulate formal models.

To integrate Cytoscape.js with OCaml, the OFLAT system uses bindings defined in the `Cytoscape.ml` module. Graph operations such as adding nodes and edges are mapped to JavaScript via `js_of_ocaml`.

5.4 Web Technologies Used in OFLAT

Understanding OFLAT's web-based architecture requires familiarity with several core technologies:

- **Web browsers** act as clients that fetch and render web content, translating HTML, CSS, and JavaScript into interactive interfaces [7].
- **HTML** provides the structural markup for web pages, defining elements like headers, inputs, and buttons [8].
- **DOM** (Document Object Model) represents the HTML document as a tree of nodes that can be dynamically modified via JavaScript [6].
- **JavaScript** enables dynamic behavior and interaction within web pages, and is essential for manipulating the DOM [9].

6

Work plan

This chapter presents the work plan for this dissertation, considering a period of six months. The objective of this work is to add support for finite state transducers (FSTs) to the OCamlFLAT/OFLAT educational system, considering three particular cases: the general case, Mealy machines, and Moore machines. A finite state transducer is a finite automaton that, in addition to recognizing words, can produce output during the recognition process. Therefore, an FST also translates an input language into an output language.

The models currently supported in OCamlFLAT/OFLAT are: finite automata, regular expressions, grammars (context-free, context-sensitive, and unrestricted), pushdown automata, and Turing machines.

6.1 Support for Finite State Transducers in the OCamlFLAT Library (8 weeks)

Several functionalities are planned at the library level. Some features might appear to directly extend those of finite automata, but the reality is more complex, for example, not all nondeterministic transducers can be determinized.

The planned functionalities include:

- **Cleaning function** – Eliminate from a transducer all unreachable and unproductive states.

- **Classification functions** – Classify a transducer as nondeterministic, deterministic, determinizable, minimal, Mealy machine, Moore machine, cleaned, etc.

- **Accept function** – Test if a transducer recognizes a given word and generates the corresponding output word. This will involve studying and instantiating the generic `accept` operation already available in the library. For words recognized by the transducer, produce a demonstration in the form of a sequence of sentential rewrites.

- **Generate function** – Determine all words recognized by a transducer up to a given maximum length, along with their corresponding translations. This will involve studying and instantiating the generic `generate` operation available in the library.
- **Conversion functions** – Convert a transducer to other model types where it makes sense; for example, convert a transducer to a finite automaton (removing the output), convert a finite automaton to a transducer (using the "identity" translation), or convert a transducer to a Turing machine (since Turing machines can represent output).
- **Integration with compound models** – Integrate transducers into the existing support for compound models in the library. This requires studying composition operations between transducers: union, concatenation, intersection, Kleene closure, etc.
- **Integration with pedagogical exercises** – Include transducers in the existing support for exercises in the OCamlFLAT library.

6.2 Support for Finite State Transducers in the OFLAT Web Application (6 weeks)

On the graphical side, the existing support for other models—particularly finite automata, which are closely related to transducers—is helpful. However, fully understanding the current implementation is non-trivial, and there are new aspects to handle, related to the fact that transducers generate output.

The tasks include:

- **Design and implement a graphical representation for transducers** – Based on graphs and tables, the two representations most commonly used in textbooks. The essential operations will be visualization and editing.
- **Expose transducer operations in the graphical interface** – Make available in the web interface the operations on transducers implemented in the OCamlFLAT library.
- **Graphically animate word translation** – Animate the process of translating a word through a transducer.
- **Maintain consistency with other models** – In general, the functionalities should be consistent with those developed for the other models in OFLAT.

6.3 Pedagogical Exercises (1 week)

Create a collection of interesting pedagogical examples and encode them as a set of exercises in the format supported by OCamlFLAT/OFLAT.

6.4 Evaluation (1 week)

- **Code correctness** – Evaluate the correctness of the new code by writing and executing a suite of unit tests during the development process.
- **Stress testing** – Determine the system's limits by creating examples that challenge it in terms of size or execution time.
- **User feedback** – Seek opinions from potential users of the system, especially regarding the graphical interface. For example, consulting instructors and students from the Computation Theory course.

6.5 Writing the Dissertation (8 weeks)

The dissertation writing should ideally progress incrementally in parallel with the other tasks. A period of two months is allocated for producing the final document. It is advisable to have a near-final version ready at least two weeks before the submission deadline to facilitate inviting an external examiner.

Bibliography

[1] C. Contributors. *Cytoscape.js*. 2024. URL: https://js.cytoscape.org/ (cit. on p. 32).

[2] Flap.js Contributors. *Flap.js Web Application*. 2024. URL: https://flapjs.github.io/FLAPJS-WebApp/ (cit. on p. 23).

[3] S. Konstantinidis et al. "Implementation of Code Properties via Transducers". In: *Implementation and Application of Automata (CIAA 2016)*. 2016. URL: https://www.dcc.fc.up.pt/~nam/publica/ciaa16.pdf (cit. on p. 20).

[4] Kyle Dickerson. *Automaton Simulator*. 2024. URL: https://automatonsimulator.com/ (cit. on p. 21).

[5] P. Linz. *An Introduction to Formal Languages and Automata*. 5th. Jones & Bartlett Learning, 2011. ISBN: 978-0-7637-7964-8 (cit. on pp. 6, 12–14).

[6] M. D. Network. *Document Object Model (DOM)*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (cit. on p. 32).

[7] M. D. Network. *How Browsers Work*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work (cit. on p. 32).

[8] M. D. Network. *HTML - HyperText Markup Language*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/HTML (cit. on p. 32).

[9] M. D. Network. *JavaScript*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (cit. on p. 32).

[10] G. van Noord and D. Gerdemann. "Finite State Transducers: Algebraic Properties and Composition". In: *Finite State Devices for Natural Language Processing*. Ed. by K. Templeton, L. Karttunen, and R. M. Kaplan. https://www.let.rug.nl/vannoord/papers/fsa/node23.html. Lecture Notes in Computer Science, Springer, 1997 (cit. on p. 16).

[11] OCaml Team. *About OCaml*. 2024. URL: https://ocaml.org/about (cit. on p. 29).

[12] Ocsigen Team. *js_of_ocaml Manual*. 2025 (cit. on p. 30).

[13] F. Project. *FAdo: Tools for Formal Languages Manipulation*. 2025 (cit. on p. 20).

[14] S. H. Rodger. *JFLAP: Java Formal Languages and Automata Package* (cit. on p. 18).

[15] Wikipedia contributors. *Chomsky hierarchy — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/wiki/Chomsky_hierarchy (cit. on p. 4).

[16] Wikipedia contributors. *Declarative Programming*. 2023 (cit. on p. 29).

[17] Wikipedia contributors. *Finite-state transducer — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/wiki/Finite-state_transducer (cit. on pp. 9, 15).