**PES Innovation Lab**

Project Report

Summer Internship 2022

# Vyaakaran

Interns

| Name | SRN |
|---|---|
| Tarun M | PES1UG20CS462 |
| Abishek Deivam | PES1UG20CS012 |
| Thejas N U | PES1UG20CS606 |

Mentors

| Name | SRN |
|---|---|
| Akash Hamirwasia | PES2201800335 |
| Hrishit Chaudhuri | PES1UG19CS187 |

PES Innovation Lab
PES University
100 Feet Ring Road,
BSK III Stage,
Bangalore - 560085

## Abstract

Vyaakaran is an educational web-tool built to help anyone understand and visualize automata theory. It has support for Regular Grammar, Context Free Grammar and powerful testing tools to help the user test automata.

We focus on extending this tool to enable simulation of a Turing Machine. We take state transitions from the user in a given syntax and pass it to a compiler to generate a graph and a tape to visualize the Turing Machine.

# Contents

# Chapter 1

# Introduction

### What is Automata theory?

Theory of automata is a theoretical branch of computer science and mathematics. It is the study of abstract machines and the computation problems that can be solved using these machines.

Automata is a machine which takes some string as input and this input goes through a finite number of states and may enter in the final state or not. Any machine that converts information into different forms using a specific repeatable process is an automaton.

### What is a Turing Machine?

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

## 1.1 Problem Statement

To create an environment to visualize and test Turing Machines.
User gives the state transitions and the input string to be tested and then Vyaakaran simulates how a Turing Machine would run it.

Automata theory is not an easy subject to understand, primarily because there are not a lot of tools built around it. The existing tools are primitive and are harder to get started with for complete beginners.

The rest of our report will go on about how we approached the problem and went about solving it.

# Chapter 2

# Literature Survey/Related Work

There are a few tools available similar to Vyaakaran:

- JFLAP - https://www.jflap.org/
  JFLAP is a package of graphical tools which can be used as an aid in learning the basic concepts of Formal Languages and Automata Theory. However this tool needs to be downloaded and has a pre-requisite of JAVA being installed.
- Turing Machine simulator - turingmachinesimulator.com
  This is an online webtool for simulating Turing Machines. However the user has no option of viewing the state transition diagram or performing test using a console.

Our initial approach was to have the user input as unrestricted grammar [1] and construct the turing machine from it. However on further research we found that this was not possible as no solid conversion exists. At best we were able to only build a turing machine unique to the input string that accepts only that string. Even this turing machine was possible only if the computer was able to determine in a 'non deterministic' manner which transitions need to be accepted and which to ignore. To replicate this non deterministic decision, we would have had to brute force through every possible decision which would have led to an exponential blowup in the space and time complexity.

Hence we changed our approach to have the user input state transitions and generate the turing machine from that.

[2] [3] [4] [5]

# Chapter 3

# Main Body

The simulator needs an input from the user in a specified format, then converts that input into a parse tree, then a graph and displays it as an interactive tape and as a state transition diagram

## 3.1 User Input

We had to design a user friendly syntax that was both easy to learn and understand

$$\begin{aligned} &S\ (\ a\ :\ x\ )\ -\!>\!-\ Q1 \\ &Q1\ (\ b\ :\ y\ )\ -\!<\!-\ Q2 \\ &Q2\ (\ c\ :\ z\ )\ -\!=\!-\ *Q3 \end{aligned}$$

We decided upon the above syntax for user input as it seemed simple enough for the user to input and understand with some simple rules -

- S represents the start state
- *Statename represents the final state
- $>, <, =$ represents the movement direction of the tape ie right , left , stay
- State names begin with capitals letters

- Symbols begin with lowercase letters

The first line essentially translates to
from start state 'S' read a symbol 'a' replace it with 'x' , move 'right' on the tape and go to next state 'Q1'

## 3.2 Compiler

[6] The Vyaakaran compiler is used to break down the user's program into a parse tree and then into a graph which is easier to display in the webtool.

### 3.2.1 Lexical Analyser

The first phase of our compiler is the lexer (lexical analyser). The lexer reads the stream of characters making up the user's input and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexer produces a token which has information like token-name and token-value which will be used by the parser. The lexer we built has various tokens for different types of input we need to handle. The tokens used are listed below:

- StartState: The character 'S', starting state of a TM
- State: Represents a state in a TM
- FinalState: The TM accepts the string
- Symbol: Any character on the input tape of the TM
- LParen: A left parenthesis '('
- RParen: A right parenthesis ')'
- Colon: A colon used as a seperator ':'
- Hyphen: Used along with Dir as a seperator
- Dir: Provides the direction the tape head moves in
- Whitespace: Skipped by the lexer
- NewLine: Denotes the end of a line and start of a new line

- Comment: Comments in a programming language

**Errors**

The lexer raises an error if any unidentified tokens are found in the input.

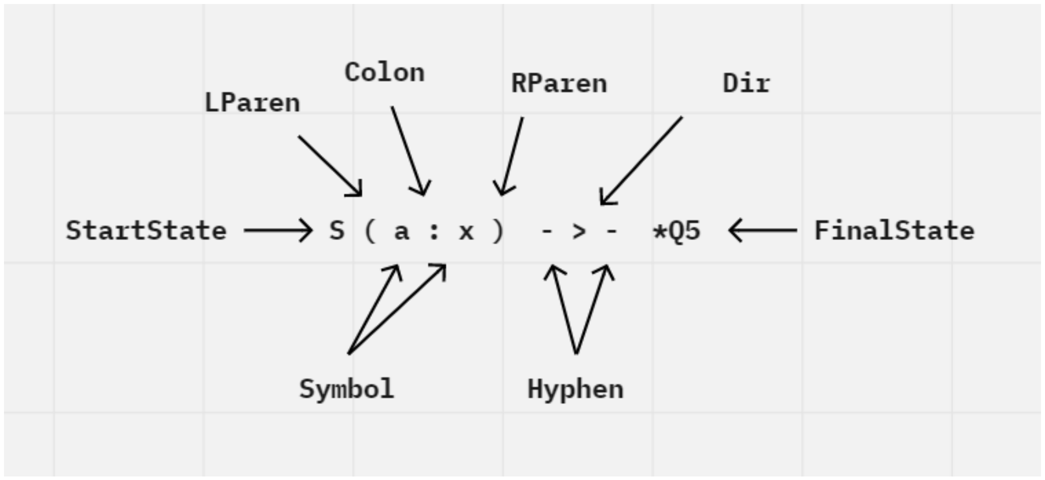


Figure 3.1: Sample Tokens

### 3.2.2 Parser

The second phase of the compiler is syntax analysis or parsing. The parser uses the list of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream which is called the parse tree. Here is our syntax's grammar.

$< START > ::= < State > < Replacement > < Move > < State > < NextLine >$
$< NextLine > ::= < NewLine > < START > \ | ""$
$< Replacement > ::= < LParen > < Symbol > < Colon > < Symbol > < RParen >$
$< Move > ::= < Hyphen > < Dir > < Hyphen >$

**Errors**

The parser raises errors if the tokens are not structured according to the grammar expected by it. This rejects input that might be a random combination of tokens which have no meaning as such.

### 3.2.3 Semantic Analyser

The semantic analyser uses the parse tree and the token list to validate if the user input in semantically consistent with our language definition

#### Graph

With the given parse tree a graph of the Turing Machine is generated. This is both used for display in the frontend and for performing graph checks in the backend. The graph checks are:

- A state is both used as a final state and a non-final state.
- Final state is reachable from the Start state.
- A Turing Machine needs to have a Start State and a single Final State.
- Two edges from a state have the same Read Symbol.

## 3.3 Hardware and Software Requirements

The client side requirement is a JavaScript-enabled browser.
The compiler is built with

- typescript ( version 4.1.3)
- parser building toolkit called Chevrotain (version 10.1.2)

The website and editor are built with -

- Vue (version 3.0.5)
- SvelteKit (version 3.44.0)
- monaco editor (version 0.22.3 )(powers VS Code)

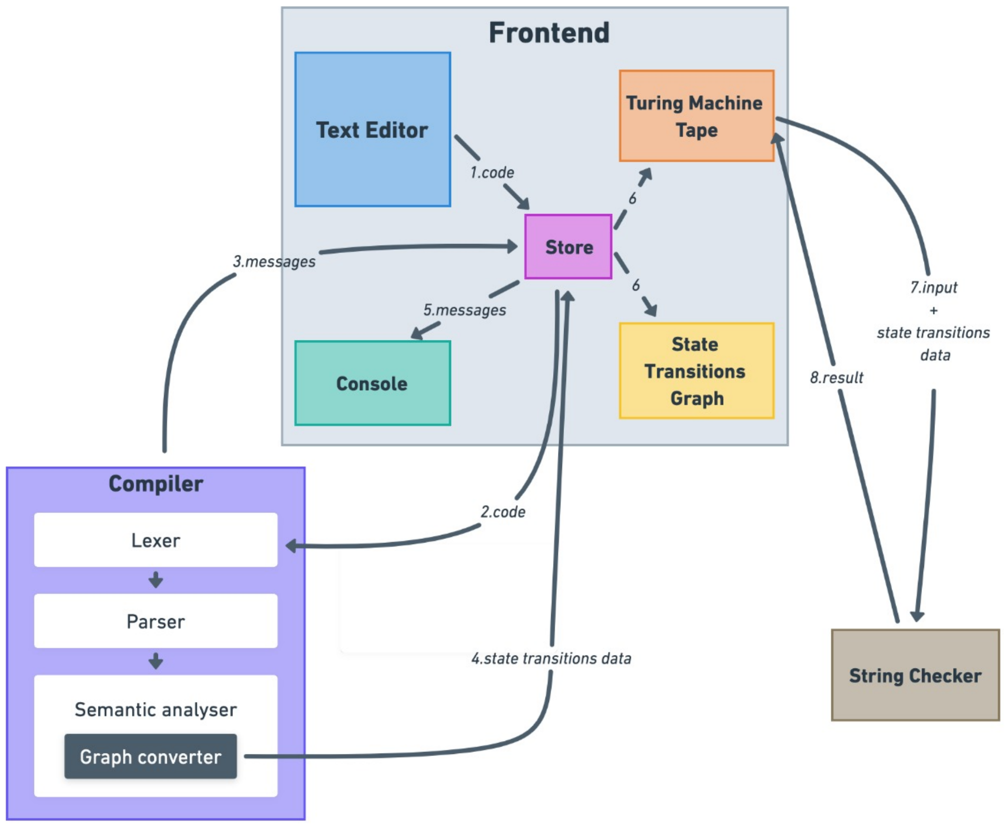
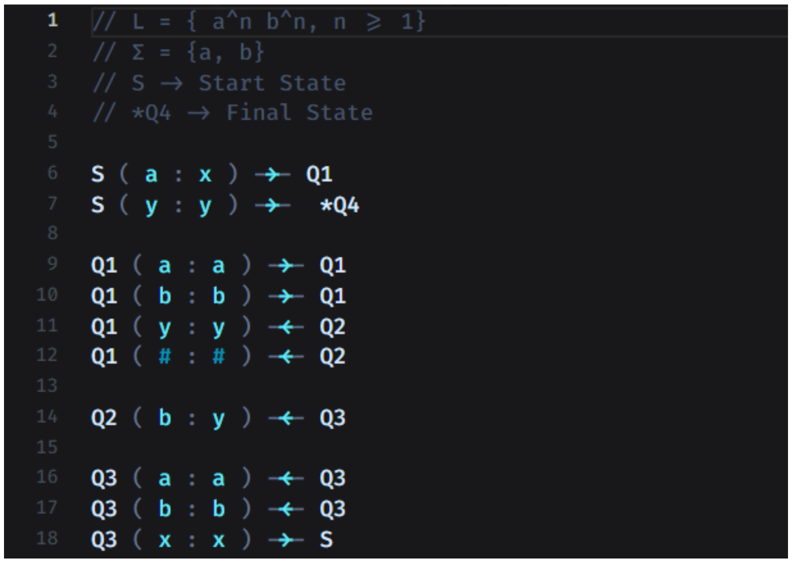


Figure 3.2: Workflow Diagram

# Chapter 4

# Results and Discussion

At this stage Vyaakaran currently supports Regular Grammar, Context Free Grammar and Turing Machine simulations. The website can be visited here



```
// L = { a^n b^n, n ⩾ 1}
// Σ = {a, b}
// S → Start State
// *Q4 → Final State

S ( a : x ) → Q1
S ( y : y ) →  *Q4

Q1 ( a : a ) → Q1
Q1 ( b : b ) → Q1
Q1 ( y : y ) ← Q2
Q1 ( # : # ) ← Q2

Q2 ( b : y ) ← Q3

Q3 ( a : a ) ← Q3
Q3 ( b : b ) ← Q3
Q3 ( x : x ) → S
```

Figure 4.1: The input given by the user

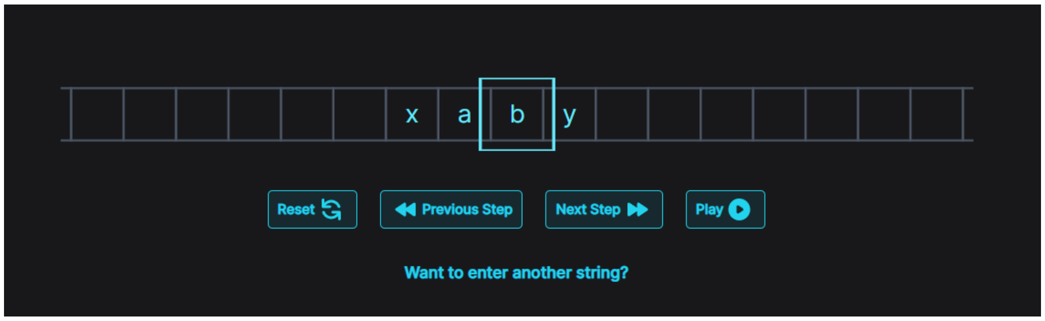


Figure 4.2: The interactive tape that the turing machine takes input from and performs operations on. The user can view a step by step process of how the turing machine accepts a string.

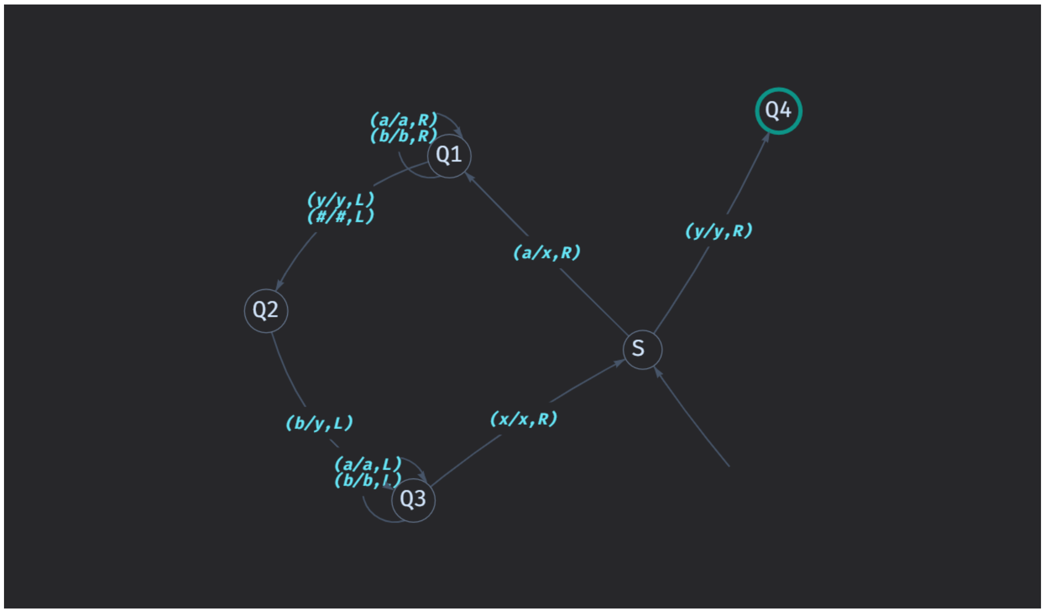
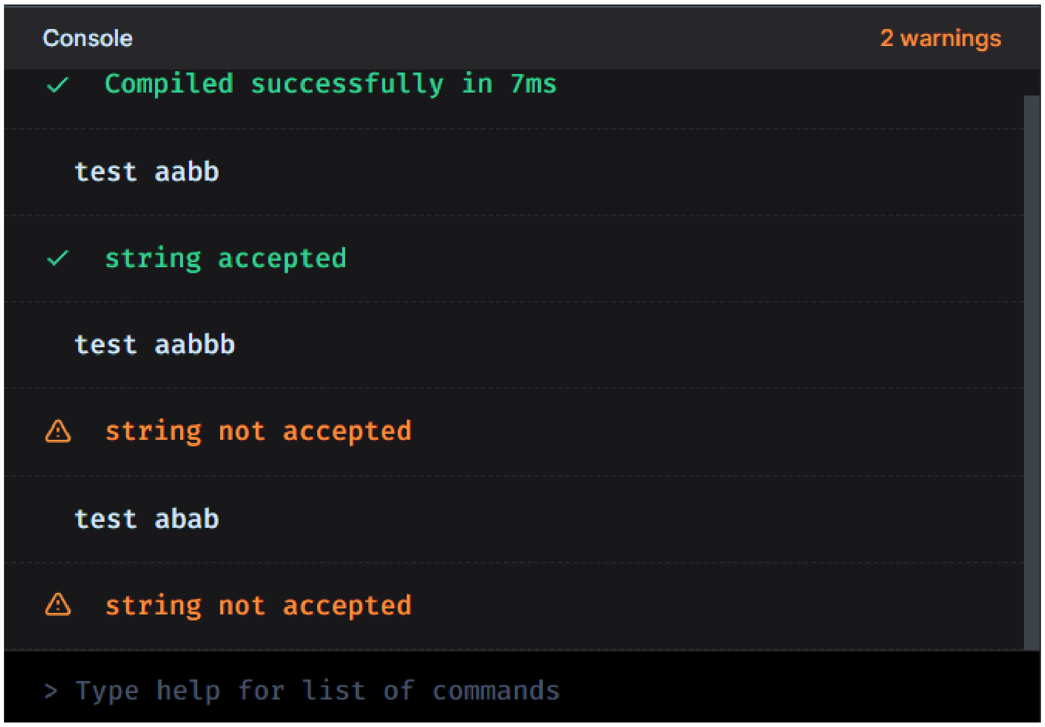


Figure 4.3: The state transition diagram that represents the turing machine in a simple format of states and transitions , this graph can also be dowloaded by the user for reference later



```
✓ Compiled successfully in 7ms
  test aabb
✓ string accepted
  test aabbb
⚠ string not accepted
  test abab
⚠ string not accepted
> Type help for list of commands
```

Figure 4.4: The console which has basic commands to compile , clear , and test strings. The compile command compiles the user input, the clear command clears the console, the test command is used to test strings through the turing machine defined by the user without the hassle of using the tape display to see whether the turning machine accepts the string.

# Chapter 5

# Conclusions and Future Work

More features and functionalities can be added. Vyakaraan is a very useful tool for anyone interested in learning automata theory, and our contribution of the turing machine simulator is a easy to use feature, with the user giving in instructions, having the instructions passed through our compiler and then generating an interactive tape and state transition graph to visualise and test the defined turing machine. Along with that we also provide an interactive console to compile and test strings.
The turing machine simulator can further be expanded by taking the user input as a pseudo code which is a much simpler way of entering instructions Multi character symbols can also be accepted , as the current system takes only single character symbols. Additionally support for multi tape turing machines can also be included.

# Bibliography

[1] Wikipedia, "Unrestricted grammar," 2022.

[2] A. Das, "Unrestricted grammar and turing machines," 2020.

[3] D. Galles, "Automata theory," 2004.

[4] J. Endrullis, "Automata theory :: Unrestricted grammars," 2019.

[5] Andrew, "Formal languages, automata and computation," 2011.

[6] T. Ægidius Mogensen, "Basics of compiler design," 2010.

# Appendix A

# Links

Website : Vyaakaran

Github Repository : Github

Chevrotain package : Chevrotain