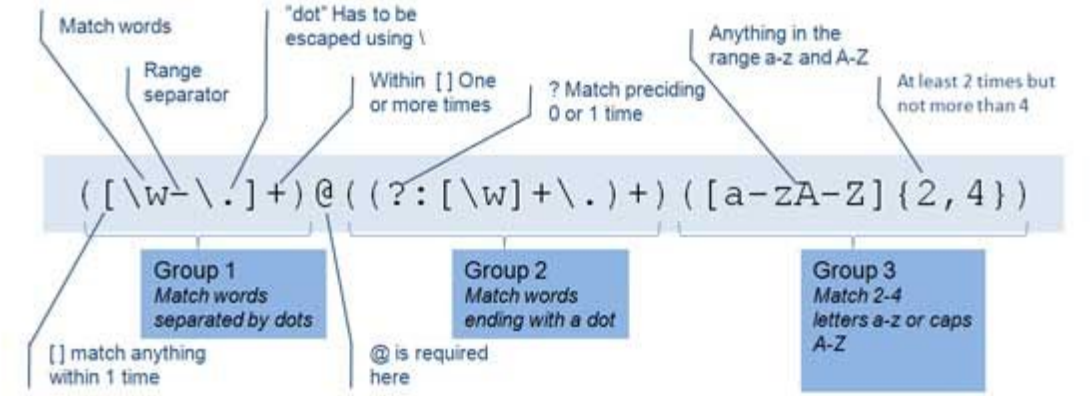
DELHI TECHNOLOGICAL UNIVERSITY

State Diagram Generator and Expression Parser

Theory of Computation (MC-304)

Anish Sachdeva
DTU/MC/2K16/013



State Diagram Generator and Expression Parser

Theory of Computation (MC-304)

16th April 2020

Anish Sachdeva
DTU/2K16/MC/13
Delhi Technological University

Acknowledgments

I would like to express my special thanks of gratitude to my teacher Prof Dr. Sangita kansal of the Mathematics Department who gave me a golden opportunity to do this wonderful project on Theory of Automata and its practical application .

I had the opportunity to learn how regular languages and deterministic automata are used in real life examples and how important regular languages are as they are used in parsers, lexical token analysers etc.

In this project I learnt how to create a deterministic automata given any regular expression, convert that automata into a recognized interface language DotScript and then further convert that DotScript into an svg image that the user can use to analyze and understand the language.

Secondly I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Index

Introduction

The project has been built on a single page web application and hosted on GitHib pages so that it can be easily viewed and used by multiple users on the internet at the same time.

The project has a textbox where the user can enter any valid regular expression and see the instantaneous output of a Deterministic Automata as soon as the expression is compiled. The expression compiling time is approximately ~100-150ms and hence the change is instantaneous to the user of the web application.

The user can view both the deterministic and non-deterministic automata using a toggle bar and both automata are compiled and generated instantaneously. The user can then also see the state transition diagram only for the finite state deterministic automata) and see the different transitions that will be made for all given symbols for all states.

The visual aid for any given regular expression helps the user better understand any regular machine and the corresponding automata that will recognize the given regular language.

The user can go one step further and after generating the automata using the given finite state machine can also check strings against that machine in linear time whether a given string is recognized by that machine or not.

This project can be used to check and parse simple strings with simple regular expressions, but can also be used for more advanced tasks. The algorithms used in generation and testing are the most efficient one's, hence we can also use this application as a lexical parser or a token analyser to parse and analyse particular tokens or strings given a very large text file by using regular expression matching.

Motivation

The first purpose of building this web application was to show to the user the synonymity of the regular expression, the finite state machine and the transition table and how they can be used to recognize the same language.

The application aims to provide the user with a visual representation for all valid regular expressions in the form of deterministic or nondeterministic automata as this helps the user (even me, the author) - better understand machines and their corresponding regular languages.

This also helps the user understand how to build finite state machines from regular expressions better. The instantaneous state transition diagram tool also shows the states and next states for all possible symbols.

The regular expression parser and string checker tool can then be used to check strings against the regular expression and the finite state machine and this text box was added to provide the user with a way to parse and check presence of strings in any given language in linear time.

Another motivation behind this application was to provide a visual interface that clearly shows the difference between the deterministic and non-deterministic finite state automata for the same regular expression (same regular language) and this difference seen visually can help users learn better how different machines with different number of states and different transitions can represent the same language.

Another motivation of building this web application was to provide future developers an interface over the DotScript Language that can be used to create finite state automatas. This web application is open source and can be viewed and extended by other developers.

Other developers can build on this application and further extend it by adding features such as support for push-down automata, compiling and lexical parsing directly using the textbox for string checking to compile entire projects in different languages.

Finite Automata

A Finite Automaton is a 5-tuple ($Q, \Sigma, \delta, q_0, F$), where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. $\delta : Q \times \Sigma \rightarrow Q$ is the **transition function**
4. $q_0 \in Q$ is the **start state,** and
5. $F \subseteq Q$ is the **set of accepted states.**

A finite automata can also be represented by it;s corresponding state transition diagram.

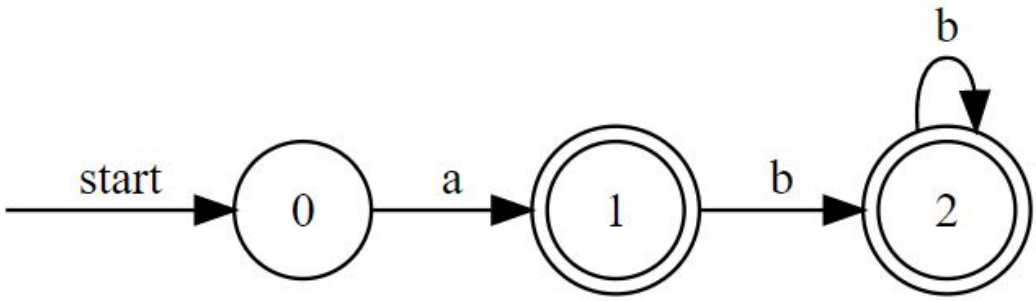


Figure 1: Finite Automata that recognizes all strings that start with exactly 1 'a' and may contain an arbitrary number of 'b' after the 'a'.

This finite state automata can also be represented with a state transition table that indicates which state it will go to after encountering a particular symbol. Φ here denotes a null state where once the automata has entered can never come out irrespective of the symbol it encountered. It is an absorbing state.

State	a	b
→ 0	1	Φ
1	Φ	2
2	Φ	2

Figure 2: The state transition table for the finite automata represented in Figure 1

Deterministic Finite State Automata (DFA)

A deterministic finite state automata is a finite automata where given a state and an encountered symbol the machine will go into only one new (or same) state and the next state can be uniquely *determined* given the symbol and current state.

This isn't true for non-deterministic machines. Some examples of finite state automata are as follows:

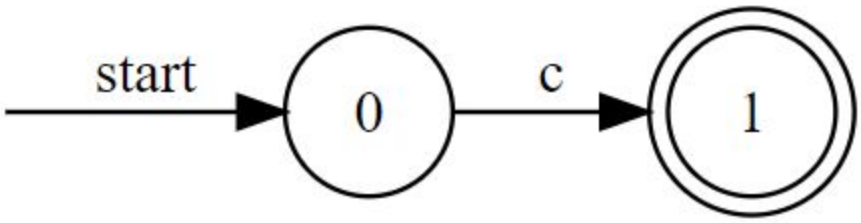


Figure 3: Deterministic Finite State Automata that recognizes only the symbol 'c'

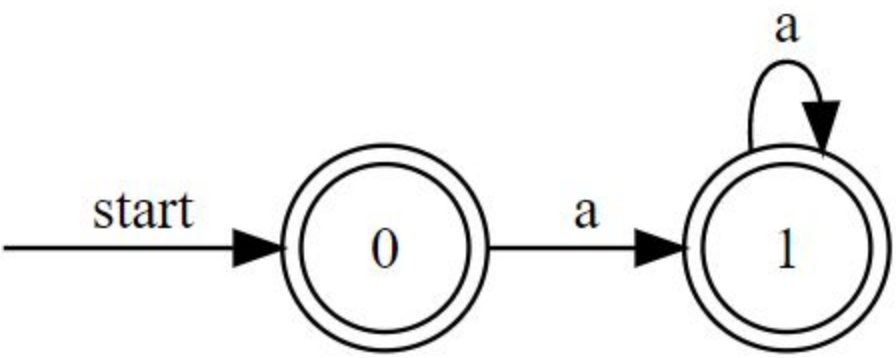


Figure 4: Deterministic Finite State Automata that recognizes the null string ε or any number of 'a'

Non Deterministic Finite Automata (NFA)

A Finite Automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. $\delta : Q \times \Sigma \rightarrow P(Q)$ is the **transition function**
4. $q_0 \in Q$ is the **start state,** and
5. $F \subseteq Q$ is the **set of accepted states.**

Where $P(Q)$ is the power set of the states. A non-deterministic finite state automata can also be represented by its corresponding state transition diagram.

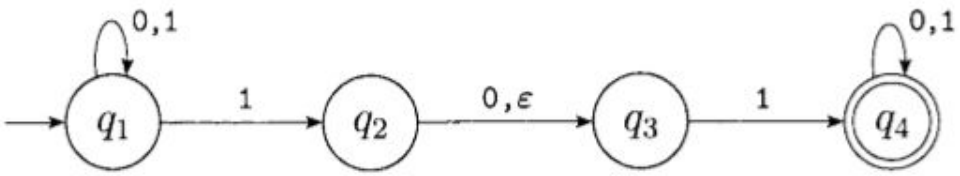


Figure 4: Non Deterministic Finite State Automata that recognizes all strings that contain the substrings as '11' or '101'

The formal definition of this automata is $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{0, 1\}$
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

,

4. q_1 is the start state
5. $F = \{q_4\}$

General Specifications

This project has been built using many technologies and multiple open source libraries.

The web application has been built on Google's angular so that there can be a unified web framework where all libraries could be included and also to make this application reactive and responsive. Using Angular allowed me to observe user managed fields and react to any change instantaneously with generating new automata on the fly.

By using angular anytime the user changes the regex column or the string checking column the result can be computed and output without the user having to reach for any button.

All code and logic has been written in Microsoft's Typescript language. Typescript is a typed superset of JavaScript and supports the basic and core EcmaScript standards. TypeScript has a very verbose codebase and by supporting types it is less error prone and will also help any future developer taking my existing code and existing it. It goes without saying that all excellent coding standards and proper coding procedures have been followed as enlisted in the Clean Code Manuscript by Uncle Bob.

The approach on creating the automata when the user enters the regular expression is as follows.

1. Firsty the regular expression is parsed and a lexical token tree is created which contains nodes for all three regular operations; Union, Concatenation and Kleene Closure. It also contains nodes for all parentheses symbols *(* and *)*.
2. Then this tree is parsed leaf by leaf and sub-tree by subtree to create an automata object. This object has the following properties and methods of the corresponding type.

```
class Automata {
 parser;
 nfa: FiniteStateMachine;
 dfa: FiniteStateMachine;
 dfaDotScript: string;
 nfaDotScript: string;
 transitionStateDiagram: TransitionStateDiagram;
 constructor(regex: string)
}
```

This object is created by passing the regular expression string and the first property this resolves is the parser property and creates a RegExParser. This parser is then used to create both the dfa and nfa, which are both of type FiniteStateMachines. These FiniteStateMachines are created with the lexical parsing algorithm as stated

above. First the regular expression string is broken down and converted to a lexical parse tree and then using that parse tree the FiniteStateAutomaton which contains information about the transitions and accept states and final states is created.

A lexical parser is a tree of the form:

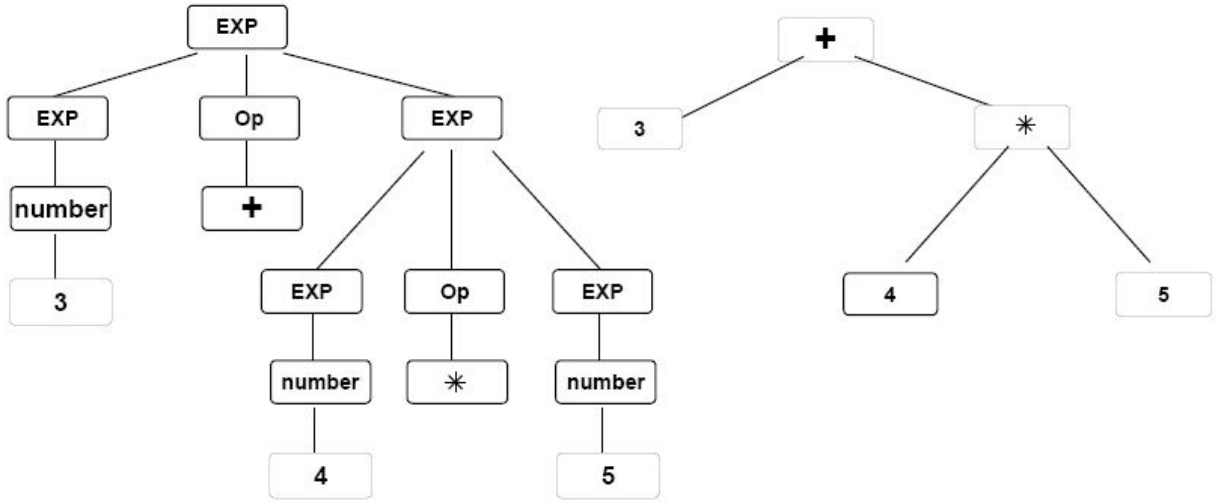


Figure 5: Example of lexical parse tree

The FiniteStateMachine object has the following structure:

```
class FiniteStateMachine {
        initialState: string;
        acceptStates: string[];
        type: 'DFA' | 'NFA';
        numOfStates: number;
        transitions;

        toDotScript(): string;
        match(text: string): boolean;
}
```

Here the toDotScript() method returns this finite state machine as valid dot script which is a recognized language and a global standard for hierarchical, tree based and graph based structures all around the world.

The FiniteStateMachine contains an **initialState** which is of type **string**. The **acceptStates** are an array of **string**. This finite State can be of 2 types and is created dynamically depending on user demand. The **transitions** is a complex object that contains all the transitions present in the `dfa` and this object is further abstracted

using the adapter class `TransitionStateDiagram` so that it can be visually represented by Angular.

3. Once the parser variable has been created **`dfa`** and **`nfa`** both and have been initialised using the lexical parse tree, now the **`transitionStateDiagram`** is created from the dfa. The **`transitionStateDiagram`** is of type `TransitionStateDiagram` and acts as an *adapter* to the **transitions** parameter so that it can be visually represented by angular and this adapter object is very important for the front facing user interface.

   The structure of `TransitionStateDiagram` is as follows:

   ```
   class TransitionStateDiagram {
           initialState: string;
           finalStates: string[];
           transitionStates: Transitions[];
           symbols = new Set<string>();
           constructor(dfa: FiniteStateMachine)
   }
   type Transitions = Map<symbol, next_state>
   ```

   The TransitionStateDiagram contains the **transitionStates** parameter that is compiled and created using the **`dfa`** that is passed in the **constructor**. The **transitionStates** is an array of Transitions objects. The Transitions object is a Map that contains the **symbol** as the key and the **next_state** as the value so that we can determine what will be the new state given a **symbol** and current state.

   The TransitionStateDiagram is created in this step and after it's successful compilation we can draw the transition table on the user facing web application. The generation of the automata image happens in an asynchronous other step.

   The asynchronous nature of the following functions

   1. `drawAutomata()`
   2. `drawTransitionTable()`
   3. `matchString() //used to see if string is recognized by given language or not`

   Ensures that the front facing application is highly responsive and can operate simultaneously as the user adds regular expression input, string input, and multiple creations can all take place together.

4. The next step is to get the dotScript file from the FiniteStateMachine objects, the **`dfa`** and **`nfa`** respectively. This is obtained by calling the toDotScript() method on the

FiniteStateMachine class. This method simply formats and serializes these objects in the universally recognized DotScript language. The DotScript format is widely used for representing trees, graphs and other connected as well as hierarchical structures.

5. The next step is to display the automata pictorially from DotScript that we have generated for both the `dfa` and the `nfa` objects. This is done using the GraphViz open source library that converts a DotScript file format into an svg (scalable vector graphic) format file. This svg file is then used to display the automata to the user.

6. The next step in the application is providing the user with the ability to check whether a string is recognized by that automata or not. This is fairly simple. When the user enters a string, the string is run on the automata using the transition table and if the string ends at a terminal state. The output is said to be valid.

7. Furthermore the transition table that is generated in the web application is done so with the help of the TransitionStateDiagram that we created earlier and as the regular expression is changed by the user, the `Automata` object is changed which triggers the transition table and svg image also to recompile and change.

This is how the application is working and the above demonstrates the flow of the application.

Running the Project on the Browser

The project has been deployed on GiHub pages and can be viewed here.

The project code has been pushed on an online repository on Github and project code can be viewed and checked out here.

The project has been published with an open source MIT license so that other people can freely view my work, extend it and also use it for their own projects.

Pull requests and suggestions can be submitted on the repository page here.

Running the Project on Your Local Machine

To run this project locally you must have the following software/packages installed on your local machine:

1. Git
2. Node
3. Angular
4. TypeScript

Angular and TypeScript can be installed after installing node on your machine. To check if node and npm have been successfully installed on your machine run the following commands:

npm --version
node --version

To add angular run the following command after adding node:

npm i -g @angular/angular-cli

To see if angular has been successfully installed run:

ng --version

To add typescript run the following command after adding node:

npm i -g typescript

Clone the repository on your machine using:

git clone https://www.githib.com/anishLearnsToCode/state-diagram-generator.git

To run locally:

cd state-diagram-generator
ng serve

This will now start running the web application on your localhost port 4200, which can be accessed by your web browser (prefer Google Chrome or Mozilla) at localhost:4200/.

Examples

Let us create the state machine for the single alphabet *a*. This can be one by simply entering *a* into the text field marked *Enter Regular Expression To generate Finite Automata*. This will generate the following finite automata.

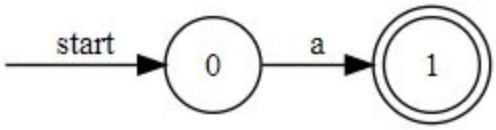


Figure 6: Deterministic Finite State Automata for the regular language that recognizes the character 'a'

A corresponding state transition diagram will also be created in the upper area panel that should look like:

State	a
→ 0	1
1	Φ

Figure 7: State Transition Diagram for the given automata in Figure 6

We can also click on the tab marked as Non-Deterministic Finite State Automata to see the visual representation of the NFA of the given regular expression. In this condition with the basic example of the regular expression that just recognizes the character *a.* The NFA will be same as in Figure 6, but if we were to take eg. the regular expression $R \rightarrow a|b$ then we will get the Deterministic Finite State machine as:

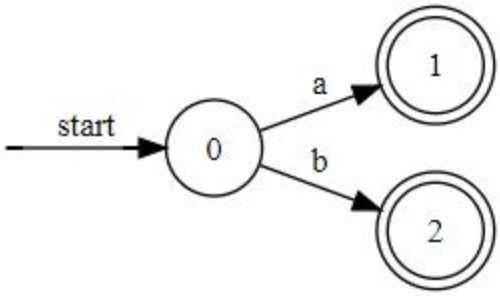


Figure 8: Deterministic Finite State Automaton for $R \rightarrow a|b$

Similarly, Clicking on the Non-Deterministic Finite State Automaton we get:

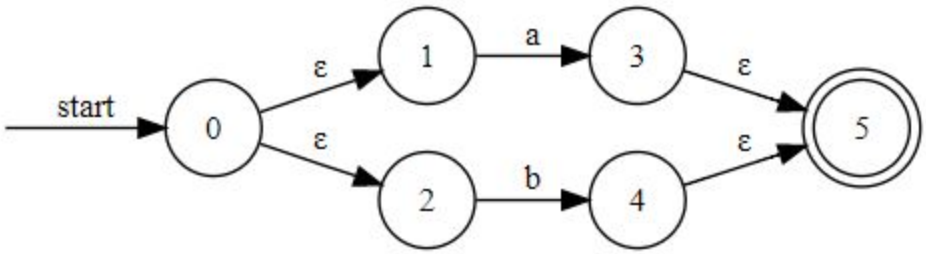


Figure 9: NFA for the regular expression $R \rightarrow a|b$

Now, we can further use the web application for checking whether a certain string is recognized by the regular language or not, or in other words whether the string is recognized by any of the given Finite State Machines or not.

We can run the given string against any machine - either deterministic or nondeterministic, but running a string against the non-deterministic will push the algorithm to split on all possible ε paths and check multiple paths for the same character which can be potentially time consuming and hurt the efficiency of the string matching.

Hence, to shorten the time taken by the algorithm we will use the optimized matching and will run it over the deterministic finite state machine. The deterministic finite state machine can take only one predetermined path for the same string.

And we also have a pre generated table of all transitions that further increases the efficiency. The running time for the string matching algorithm over the deterministic finite state automaton is $O(n)$ where n is the length of the string.

We can enter a string to check as follows and it will also display an output whether it belongs to the regular expression or not.

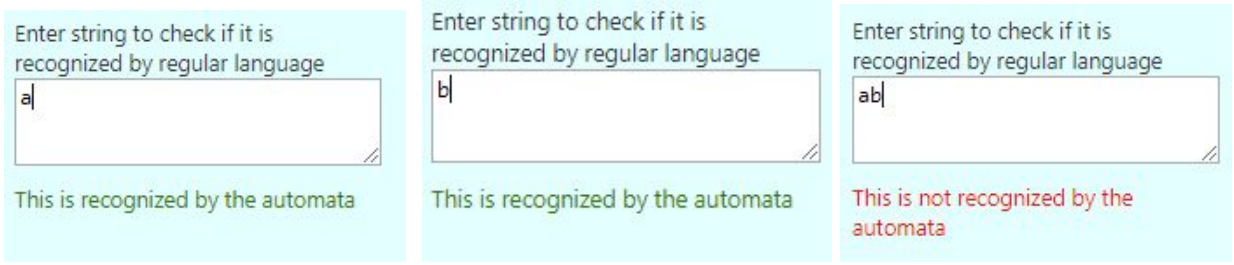


Figure 10: String matching to see whether it belongs to a language or not

Conclusion

Using the web application **Automata Creator and Checker**, users can create visual representations of deterministic and non-deterministic automata from any valid regular expression.

This visual representation is generated instantaneously and this application can be used for the aid in creation, understanding and even educational purposes to see how a finite state automaton is formed.

This web application can also further be used to create transition state tables for a regular expression on the fly. Where automata is very visual and helps us in our understanding the transition table is something that is efficient and is used by us when we are solving problems on pen and paper and the transition tables are even used internally in computers and in algorithms involving regular expressions and automata.

The transition tables can be hashed by a computer and hence they can be traversed over very efficiently. Even the inbuilt implementation of the string matching in this web application uses the transition table, which makes string matching efficient in this implementation.

In conclusion, this web application completely covers the visual concept of a finite state automaton and also adds a generator for transition table and string matching functionality which can be accessed by anyone anywhere in the world with an internet connection.

Bibliography

1. Git
2. GitHub
3. Angular
4. Node
5. Npm
6. State-diagram-generator [GitHub - Repository]
7. Introduction to Theory of Computing [MIT Press]
8. Angular Hosting on Github Pages [Telerik]
9. gh-pages [npm]
10. Introduction to Theory of Computation by Michael Sipser [Cengage Learning]
11. Deterministic Finite State Automaton [Wikipedia]
12. Clean Code by Robert C. Martin (Uncle Bob) [Amazon]
13. GraphViz
14. DotScript [Wikipedia]