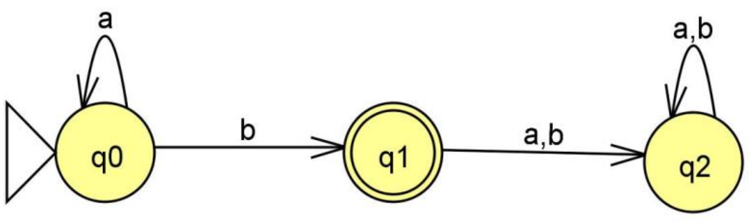
Simulators for formal languages, automata and theory of computation with focus on JFLAP



Tobias Fransson
tfn09001@student.mdh.se

Institutionen för Datavetenskap och Elektronik (IDE)
Handledare och examinator: Gordana Dodig-Crnkovic

Västerås, 2013

ABSTRACT

This report discusses simulators in automata theory and which one should be best for use in laboratory assignments. Currently, the Formal Languages, Automata and Theory of Computation course (FABER) at Mälardalen University uses the JFLAP simulator for extra exercises.

To see if any other simulators would be useful either along with JFLAP or standalone, tests were made with nine programs that are able to graphically simulate automata and formal languages. This thesis work started by making an overview of simulators currently available. After the reviews it has become clear to the author that JFLAP is the best choice for majority of cases. JFLAP is also the most popular simulator in automata theory courses worldwide.

To support the use of JFLAP for the course a manual and course assignments are created to help the student to getting started with JFLAP. The assignments are expected to replace the current material in the FABER course and to help the uninitiated user to get more out of JFLAP.

SUMMARY

JFLAP is the currently used simulator in the Formal languages, Automata and Theory of Computation course (FABER) at Mälardalen University. The tests are made with nine programs including JFLAP to see which one that met all our requirements. Every simulator was tested with a set of test cases and discussions were made about if the tested automata simulator met given requirements. The manual and assignments are created for JFLAP which is found to be the best simulator to be used in the FABER course at MDH.

PREFACE

This report is a B.Sc thesis work in computer science. It was written at the School of Innovation, Design and Engineering (IDT) at Mälardalen University.

Västerås, 2013
Tobias Fransson

NOMENCLATURE

Alphabet – a set of symbols. Example: {a,b} or {0,1}.

String – a sequence set over an alphabet
Example: aabbabb

Languages – is a set of strings
Example: L = {$a^n b^n$; n>0}

Grammar – a set of rules that formulate strings in a language.
Example: a grammar over the alphabet {a,b}

$$S \rightarrow aSb$$
$$S \rightarrow \lambda$$

Specifies the language: L = { λ, ab , aabb , aaabbb , .. } = {$a^n b^n$; $n \geq 0$}

FA – Finite Automaton – is a model of computation that consists of a (finite number of) states and transitions between those states. It reads an input string and after number of state transitions either accepts or rejects the string.

DFA – **Deterministic finite automaton** - is a finite state machine that produces a unique computation for each input string. From each state on a given input symbol the automaton may proceed into one possible next state.

NFA – Nondeterministic finite automaton - each state can have more than one transition with the same input value. Lambda transitions can also be used which enables the automaton to move from one state to another without consuming any input.

PDA – Pushdown Automaton – is an automaton that has a stack where the push and pop function is used. Because of this some iterative patterns can be evaluated. For example:

$$A \rightarrow (A)$$
$$A \rightarrow ()$$

Which accepts L = {(), (()), ((())), ..}.

TM - Turing Machine – is a more complex automaton that has a tape that can be read and written to. The head points at the beginning of the string when the TM executes. Every transition has three variables which are read, write and the direction.

CONTENTS

1. INTRODUCTION 6

1.1 BACKGROUND 6
1.2 OBJECTIVE 6
1.3 PROBLEM FORMULATION 6
1.4 LIMITATIONS 6

2. METHOD(S) 7

2.1 PREVIOUS WORK 7
2.2 COLLECTION OF SIMULATORS 8
2.3 REVIEW OF SIMULATORS 8
2.4 THE MANUAL 9
2.5 THE ASSIGNMENTS 9

3. REVIEW RESULTS 10

3.1 SIMULATOR REVIEWS AND CONCLUSIONS 10
Simulator: Automata Editor 10
Conclusion: Automata Editor 12
Simulator: Automata Editor by Max99x 13
Conclusion: Automata Editor by Max99x 15
Simulator: Automaton Simulator 16
Conclusion: Automaton Simulator 17
Simulator: JFAST 18
Conclusion: JFAST 20
Simulator: JFLAP 21
Conclusion: JFLAP 23
Simulator: PetC 24
Conclusion: PetC 25
Simulator: Tuatara Turing Machine Simulator 26
Conclusion: Tuatara Turing Machine Simulator 27
Simulator: Turing Machine Simulator 28
Conclusion: Turing Machine Simulator 29
Simulator: Visual Automata Simulator 30
Conclusion: Visual Automata Simulator 32
3.2 SIMULATOR ANALYSIS DISCUSSION 33

4. CONCLUSIONS ON THE CHOICE OF SIMULATOR 34

5. FUTURE WORK 35

6. REFERENCES 35

7. APPENDICES 37

7.1 APPENDIX 1 37
7.2 APPENDIX 2 37

8. MANUAL AND EXERCISES 38

8.1 APPENDIX 1. MANUAL FOR JFLAP SIMULATOR USE IN THE FORMAL LANGUAGES, AUTOMATA AND THEORY OF COMPUTATION COURSE ....
8.2 APPENDIX 2. ASSIGNMENTS ....

1. INTRODUCTION

1.1 Background

Learning about formal languages, automata and theory of computation often involves long and tedious constructions of automata, grammars, derivations of strings and test running of automata that all are error prone. At Mälardalen University the Formal languages, Automata and Theory of Computation course (FABER) has been using a simulator called JFLAP [4] for the course exercises. Using this automata simulator program helps students with experimenting with automata and grammars and getting a better idea about how they work. It is also much easier to control and compare solutions.

1.2 Objective

The objective of the present work is to test a collection of automata simulators that is available on the internet. Every simulator should be able to simulate all or some of Finite Automata, Pushdown Automata and/or Turing machines. The program should be easy to use and have documentation well enough for anyone familiar with automata theory. The related objective is to create assignments for the course that uses JFLAP or a preferred simulator. Also a manual for the preferred simulator should be created that the students can use during the course.

1.3 Problem formulation

There are many applications found on the internet that can simulate automata but are they any better than the currently used JFLAP? Which of these are better suited in laboratory assignments with automata? How to create good laboratory assignments by using JFLAP for use in a Formal Languages, Automata and Theory of Computation course?

1.4 Limitations

This report presents the results of the tests of simulators that handle either one or more of DFA, NFA, PDA and/or Turing machine types of automata. Tests have not been done with any frameworks or libraries that handle automata or state machines. Requirement was for the simulator to have at least some sort of graphical user interface.

2. METHOD(S)

2.1 Previous Work

Similar tests have been done in an earlier exam report was written in 2006, when a simulator called A5 was used in the FABER course at MDH. The review came to the conclusion that JFLAP was much better suited for use in the course assignments. The question was if some new simulators appeared in the meantime that could be used in the course.

The creators of the JFLAP simulator are Professor Susan H. Rodger of Duke University together with her students. JFLAP is the most successful simulator worldwide with over 25.000 downloads since 2003 [14].

Simulators like JFLAP can be called a multipurpose automata simulator because of the variety of automata they can handle. While there are other simulators specified for one or several types of automata, it could be a good idea to consider using more than one simulator during a course, according to [17].

Currently the *Activities CD-ROM for JFLAP* is available from the Linz book [12] that contains several assignments with JFLAP files included. The assignments are easy to begin with and get more in-depth later on. There is a course called Introduction to Formal Languages and Automata at Mississippi State University that have homework using both traditional methods and JFLAP [10]. These assignments are taken from the Linz book which is the same as the book used in the FABER course.

There are several ways to teach automata by visualization that is useful for students. One way is so called hypertext books [13]. These were HTML pages that have active learning applets which the students are encouraged to step through on their own.

Majority of simulators found on the internet are have no graphical interface but they are frameworks that either have the necessary algorithms (such as conversion between automata types, minimization etc.) or drawing tools. One of them is FAgoo which contains graph drawing algorithms for arranging finite automata [15]. GraphViz is used on some of our test simulators and is very extensive graph viewing software that supports most type of graphs [16].

2.2 Collection of simulators

Most of the simulators are found using the Google Search engine with the keywords: “automata automaton Turing Machine Pushdown grammar simulator sim editor state graph”. Only exception is JFLAP which was known beforehand from the FABER course.

2.3 Review of Simulators

To select the best and most usable simulator tests have to be made with each of them. The best simulator is the one easiest to use and have enough features to aid the user with. By creating different automata types that the current program supports we can get an image of how helpful the program really is. Test cases were created for different automata to test each simulator (see Table 1).

DFA	1. a*b 2. aWaaWa; W = {a,b}* 3. aa*(a+b)a
NFA	1. a^+ab^*a 2. (001*)|(010*) 3. aa(a+b)*ba
PDA	1. A = (A), A = (), A = e 2. S → AB, A → aaA | a, B → bBa | bb 3. S → AB, A → aAb | ab, B → aaB|a
Turing Machine	1. Duplicate the number of ones, Ex 1111 → 1111 1111 2. L = $\{a^n b^m\}$, n > m > 0 3. L = $\{a^n b^n c^n\}$, n > 0

Table 1 Test cases for different types of automata

Each simulator is rated within six different categories. These categories are discussed separately for each simulator and a score is given for each simulator. These criteria gets a grade from 1 to 5 where 1 is the worst and 5 is the best (see Table 2).

Functionality: What functions are there and how they help the user. How many types of automata supported.
Tools: How do the different tools available help the user? Mainly aims the automata debug features.
User Friendliness: Is the application user friendly?
Layout/Design: How the application is structured and its graphical quality.
Compatibility: Do the application work on any PC? Is it difficult to start?
Documentation: Is there a manual or documentation available?

Functionality, documentation and compatibility will discuss which features (ex. platform independent) each simulator has. Tools, user friendliness and layout will be more what the author thinks about each simulator. Thus the review will reflect both what the author thinks about each program and its actual features. To get a grade four in Functionality a simulator should handle all automata types plus regular expressions and grammar, plus conversion from regular expression or grammar to NFA’s. Determination and minimization algorithms are also considered. A grade five should be more than that.

Criteria	Grade 1	Grade 2	Grade 3	Grade 4	Grade 5
Functionality	No or few functions available	Some functions available	Most functions available	All necessary functions available	More than all necessary functions available
Tools	Hard to use Debug features not helpful No output	Hard to use Debug less helpful Minor output	Some difficulties to use Partially helpful Some output	Easy to use Debug is helpful Detailed output	Easy to use Debug is very helpful Detailed output
User Friendly	Not user friendly	Less user friendly	Partially user friendly	User friendly	Very user friendly
Layout Design	Inconsistent layout Hard to navigate Poor graphical quality	Inconsistent layout Insufficient graphical quality	Consistent layout Average graphical quality	Consistent layout Easy to navigate Good graphical quality	Consistent layout Easy to navigate Best graphical quality
Compatibility	OS dependent, Difficult to begin with	Some cross platform capability	Cross platform Easy to begin on at least one OS	Cross platform Easy to start on most OS's	Cross platform Easy to begin on any OS
Documentation	No documentation	Minor documentation	Manual or guide showing how the tool works	At least a manual covering most issues	Complete documentation and/or more

Table 2 Grading criteria

Simulators that prove difficult to use do not get tested in detail. Further inspection will be made if they are useful in any other way by discussing above categories. After the review the best suitable simulator will be presented and discussion about how these programs are used either in assignments or the manual is also presented.

2.4 The Manual

To make students more comfortable with our preferred simulator, a manual is made. This manual explains most of the simulators features and how to use them correctly. The manual is very straightforward and easy to pick up if the user should be stuck at any point.

2.5 The assignments

The assignments based on the preferred simulator program are prepared. It is presupposed that the students should be familiar with the basics of formal languages and automata theory. These assignments should be enough to get the students a kick start in learning how to use the simulator and be more comfortable in creating different types of automata.

3. REVIEW RESULTS

3.1 Simulator Reviews and conclusions

Simulator: Automata Editor

Home Page: http://automataeditor.sourceforge.net/
Current Version: v 2.0

Description

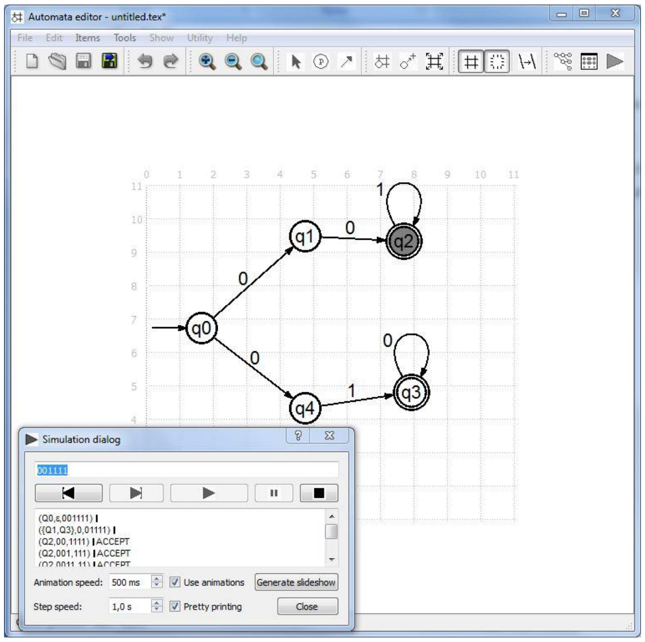
Automata Editor is an automata editor that utilizes a format called VauCanSon-G which is a LaTeX package [5].

Features

- Create and edit DFA and NFA
- Determination and Minimization

Review

Automata Editor always starts in the editor with no distinction for if either DFA or NFA is used. Here you can add states and transitions from the toolbar. Options like undo and save alongside with graphical fidelity settings are available from the toolbar as well.



Automata editor - untitled.tex*
File Edit Items Tools Show Utility Help
q0
q1
q2
q3
q4
0
0
0
1
1
0
Simulation dialog
Generate slideshow
Close

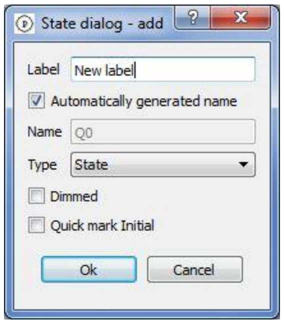
001111

```
(Q0,ε,001111) |
({Q1,Q3},0,01111) |
(Q2,00,1111) |ACCEPT
(Q2,001,111) |ACCEPT
(Q2,0011,11) |ACCEPT
```

Animation speed: 500 ms ☑ Use animations
Step speed: 1,0 s ☑ Pretty printing

Figure 1 Automata Editor with NFA (001)|(010*)*

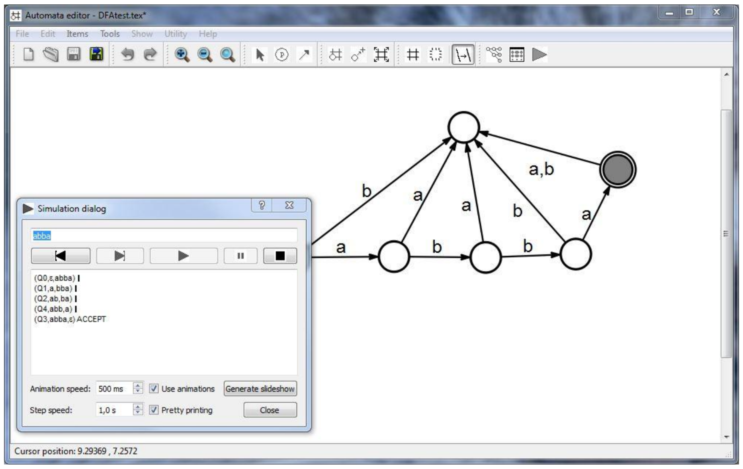
Each time you add a state or transition you get a dialog where you can set some settings for this item. You can safely pass this one and set them later by left click then select “Edit”. The “Quick Mark Initial” sets the state as an initial state.



Label: New label
[✓] Automatically generated name
Name: Q0
Type: State
[ ] Dimmed
[ ] Quick mark Initial

Figure 2 Settings for added states in Automata Editor

The debug window run in front of the editor and gives debug output and the ability to pause the execution. The currently active state is rendered in grey which changes during execution. The debug or “simulation dialog” prints out every step which is very easy to understand.



abba

```
(Q0,ε,abba) ⊢
(Q1,a,bba) ⊢
(Q2,ab,ba) ⊢
(Q4,abb,a) ⊢
(Q3,abba,ε) ACCEPT
```

Animation speed: 500 ms [✓] Use animations
Step speed: 1,0 s [✓] Pretty printing

Cursor position: 9.29369 , 7.2572

Figure 3 Automata Editor with debug output

During tests with NFA I could not find how to create lambda transitions. Other than that creating NFA was as easy as DFA and the debug tool did run correctly, which makes the simulator able to process some NFA’s.

Conclusion: Automata Editor

Functionality

The application includes automata determination and minimization. The program lacks some useful features. There is no conversion from automata to grammar or regular expressions.

User Friendliness

Making automata is easy and straightforward. There are however some negative features. I do not like the popup settings that show up every time you add an item. I got into problems creating usual NFA’s with lambda transitions, I cannot add lambda transitions and using “e” or “eps” as epsilon did not work either which is confusing.

Tools

Debugging works well, you see what happens and every transition is animated with a moving dot where the active state has a different color. The debug output is well printed with the mathematical notation similar to a transition table.

Design

The program could be simpler. The toolbar have a couple of functions that changes graphical setting like anti-aliasing or changing the background grid which would be better hidden in a graphical options menu. Instead there could be a menu there to set the values for a selected state. The good is that the debug tool is well made and easy to use.

Documentation

The documentation is sparse and lacks a detailed manual.

Compatibility

The source code uses a Nokia Qt based user interface and Graphviz for automata graphics.
A Windows binary and source code is available from the home page. Getting started with the source code might get problematic for some students and without proper instructions it is too time consuming.

Simulator: Automata Editor by Max99x

Home Page: http://max99x.com/school/automata-editor
Current Version: 1.0

Description

Automata Editor is a simulator created in the language Python. It is created by a former student as a project in a Theory of Computation course [8].

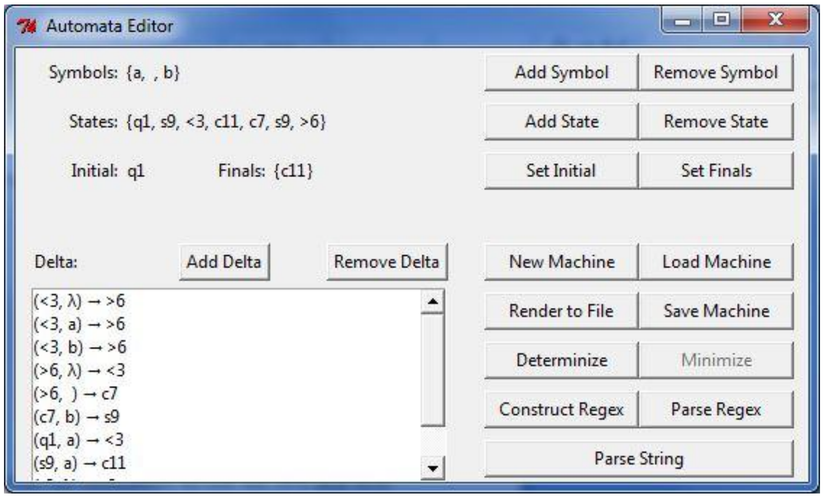
Features

- Create and edit DFA and NFA
- Convert NFA → DFA
- Convert RE ↔ NFA
- Determination and Minimization

This editor does not have a proper graphical interface like JFLAP, which means that you have to programmatically insert every item into your automaton.

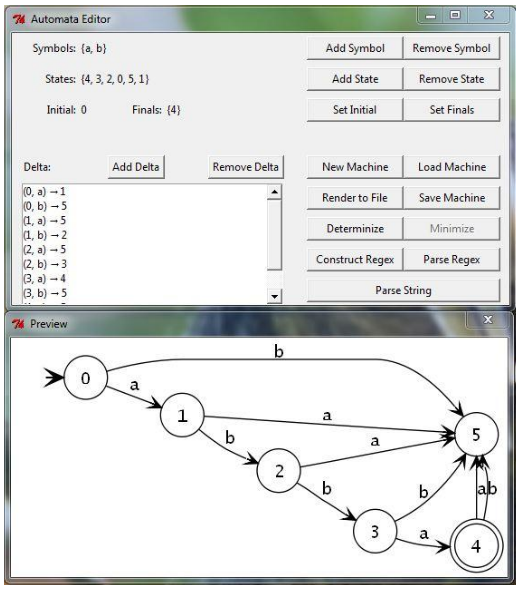
Review

With Automata Editor you do not get a graphical interface to drag and drop states around. You have to specify each item trough adding them or create a regular expression. After a proper automaton is specified Automata Editor should then render your automata in a separate "Preview window". Add and remove symbols means which characters the automata should use. The Add and Remove State handles states, where you can set the initial and final states in the Set Initial and Set Finals buttons. The Add and Remove Delta handles the transitions. Except for the "Preview" window the current automaton is represented in a list in the main window.



```
Symbols: {a, , b}
States: {q1, s9, <3, c11, c7, s9, >6}
Initial: q1          Finals: {c11}
Delta:
(<3, λ) → >6
(<3, a) → >6
(<3, b) → >6
(>6, λ) → <3
(>6, ) → c7
(c7, b) → s9
(q1, a) → <3
(s9, a) → c11
```

Figure 4 The main window in Automata Editor



Symbols: {a, b}

States: {4, 3, 2, 0, 5, 1}

Initial: 0 Finals: {4}

Delta:

(0, a) → 1
(0, b) → 5
(1, a) → 5
(1, b) → 2
(2, a) → 5
(2, b) → 3
(3, a) → 4
(3, b) → 5

Figure 5 Automata Editor with DFA accepting the string ‘abba’

By pressing “Parse Regex” you can test the automata with a single input. You get either an accepted or rejected massage after the test is complete.

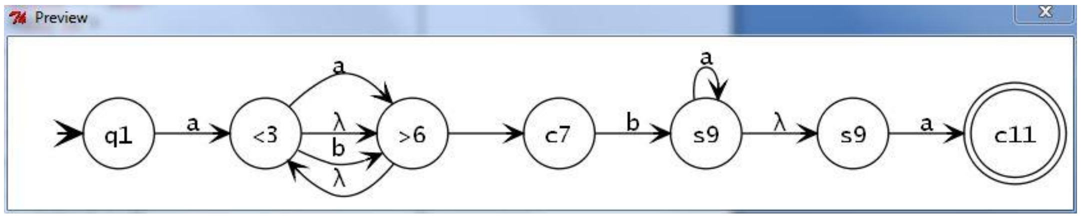


*Figure 6 Parsed from a(a+b)*ba*a*

Creating automata without planning is not a good idea. You might need to rethink your design on a pen and paper. It is easier if you have your automata ready and just input all the states and transitions. This makes experimentation hard because the interface is so restrictive. It is faster to create regular expressions that you input using the “Parse Regex” button. The program does not remember used expressions so you have to remember them yourself. The “Preview” windows do manage to handle most automata and makes them easy to follow and read. Though there is no way to graphically manipulate the result in this preview window. Automata Editor handles NFA just as well as DFA. But do note that Automata Simulator do not tell DFA and NFA apart in any way.

Conclusion: Automata Editor by Max99x

Functionality

The application can convert regular expressions to automata.
It lacks a proper editor window to graphically manipulate your automata.
Determination and Minimization algorithms are available.

Tools

It is possible to debug strings one at a time. Lack of any “step by state” feature makes debugging harder. You either get an “Accepted” or “Rejected” message after the input string have been used.

User Friendliness

By creating automata you have to specify every state with transitions in a list of states and transitions, which takes time and is easy to lose track. Though when the work is done the automata is rendered nicely and transitions are separated so they do not overlap each other. Using regular expressions to generate finite automata is much more efficient but the user must know how to create regular expressions that are accepted by the program.

Design

The layout is simple and easy to understand but do not help user who is unfamiliar with automata.

Documentation

The documentation is slim. The application comes with example automata that the user can look at. There are no instructions on how to construct regular expressions.

Compatibility

Automata Editor uses Python and the GraphViz library. I was not successful in running from source code but there is a Windows binary available which worked well. Without a proper editor students may have trouble understanding how this program works.

Simulator: Automaton Simulator

Home Page: http://ozark.hendrix.edu/~burch/proj/autosim/
Current Version: 1.2

Description

Automaton Simulator is a simple and open source automata simulator.
It is created by Dr Carl Burch at Hendrix University [6].

Features

- Create and edit DFA, NFA, PDA and Turing machines

Review

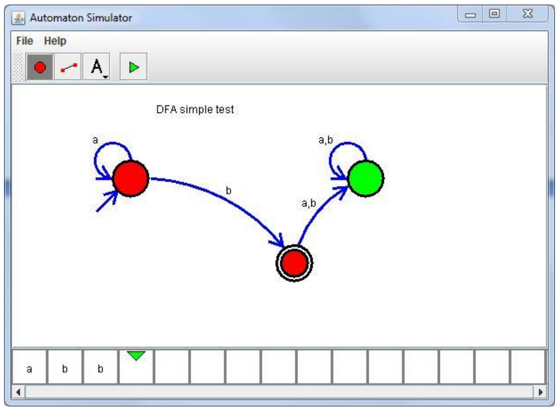


*Figure 7 Automata Simulator with the language a*a*

Automaton Simulator directly puts you in the editor window with DFA currently used. Here you can edit your automata easily by selecting either the state or transition button and simply add them to the workspace using the mouse. When pressing the run button you can write your input in the tape below to automatically move one state. The active state is green and an animated dot run through the transitions while jumping states.

Creating DFA, NFA, PDA or Turing machines is fast and simple. Edit different types of automata might differ some, like debugging a Turing machine because you need to edit the tape before you start debugging. The application might be little to simple and gives no features whatsoever. One important feature missing is a better set of alphabet, only ‘a’ and ‘b' seems to be available.

Conclusion: Automaton Simulator

Functionality

There seem to be few functions included with Automaton Simulator. Among other there are lack of conversion between regular expressions and FA and no determination and minimization.

Tools

The debugging is very good, when you press a letter into the tape, the states change immediately to green when they are active. This makes it easy to follow what is happening. There is no transition table that shows all previous steps.

User Friendliness

Creating finite automata is straightforward and simple. The downside is that it doesn't support using different languages, you only have alphabet with 'a' and 'b'. It is sometimes hard to know what to do or not, like whenever you can add something onto the tape. Sometimes it works but if there is anything wrong with your automata, the tape will not respond. The program does not tell what certain settings mean, like when adding values to a PDA transition there are no hints if you have your mouse over push or pop. The application lacks a selection tool which makes it very easy to create new states by mistake.

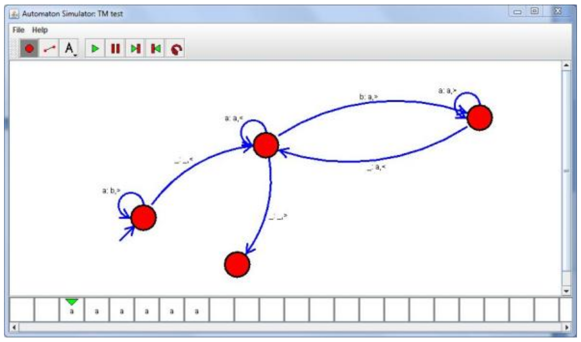


Figure 8 Automaton Simulator with Turing machine that duplicates numbers

Design

The layout of the application is simple and straightforward.

Documentation

The home website has a documentation page which covers the basics of the program.

Compatibility

The simulator runs with Java 1.3 and is cross platform. The application is very simple and debugging is very easy to use which is useful for most assignments.

Simulator: JFAST

Home Page: http://sourceforge.net/projects/jfast-fsm-sim/
Current Version: 1.3

Description

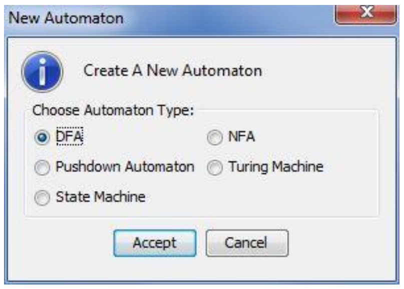
The simulator is written by Timothy M. White that first made this simulator as his B.S in computer science at Villanova University. By that time it only did handle DFA but White has been developing the program since then and now the application has more features [2].

Features

- Supports DFA, NFA, PDA, TM and State Machines
- Save image to PNG or JPEG

Review

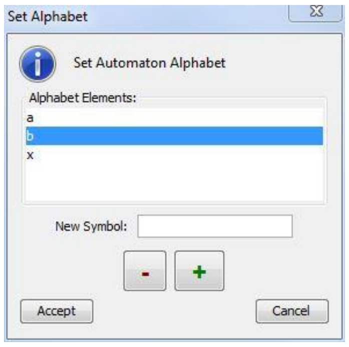
You get straight to the editor window once the program starts. By using “new” in the file menu you can select the automata type you prefer.



Choose Automaton Type:
[✓ DFA] NFA
Pushdown Automaton Turing Machine
State Machine

Figure 9 The selection menu in JFAST

You have to specify the alphabet before you create and run your automata. The button “Set input Alphabet” specifies the alphabet. You press any letter and select the plus button which adds that letter to the alphabet set used in the current automaton.



Alphabet Elements:
a
b
x

New Symbol:

Figure 10 Alphabet setup in JFAST

The state and transition buttons in the upper toolbox creates states and transitions. When creating a transition an options menu shows up to let the user set input characters from the alphabet. Every item in the workspace can be changed by right clicking them. Initial states are blue and final states are green.

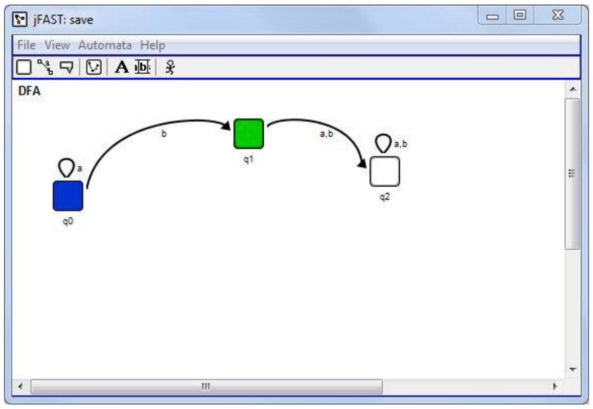


*Figure 11 JFAST with the language a*a*

You test your automata with the *Run* button at the right. It starts the simulator mode in a full screen overlay, unfortunately that mode made the program unresponsive using Windows 7. I tried creating my first automaton I ran into several problems. Transitions kept disappearing and the automata did not accept my input without explanation. The transitions also curve too much they often end up in a mess.

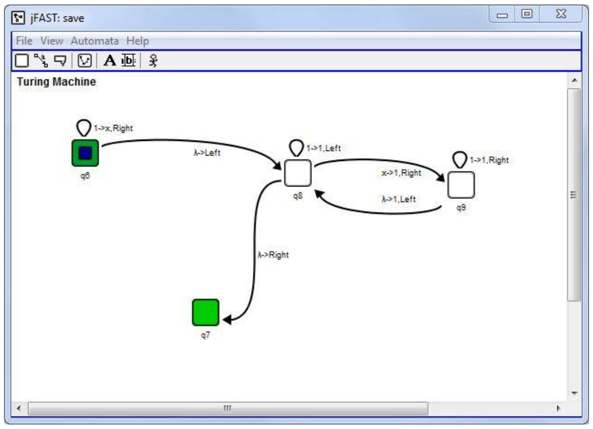


Figure 12 JFAST with Turing machine that duplicates numbers

Since there where so many problems, it is hard to do larger projects using JFAST. Creating PDA and Turing machines were also harder to do than simple FA.

Conclusion: JFAST

Functionality

No minimization or determination or automata conversion functions are implemented in JFAST.

Tools

The debugging part crashed for me for some reason, which made it hard to test debugging properly.

User Friendliness

JFAST could be good if it did not have all these bugs. Transitions disappeared and reappeared at random. You must have a start and end state and specify your alphabet to run anything or else a warning message is shown. To set the alphabet you need to add each letter separately, which I didn't realize the first time. I also had problems running strings with my creations which other automata simulators accepted easily. Like most other simulators the interface is simplified, but creating automata is often tedious and time consuming.

Design

Graphics is minimalistic and good looking. For some reason states are box-shaped and not circles like in other simulators. The transitions curve themselves and it is hard to make them look good.

Documentation

There is a textual tutorial on the home page with basic information.

Compatibility

Uses Java and is cross platform. But it did crash for me while testing using Windows 7.
The program is available on the *Sourceforge* webpage which have not been updated since 2009.

Simulator: JFLAP

Home Page: http://www.jflap.org/
Current Version: Version 7.0

Description

JFLAP is created by Susan Rogers and her students at Duke University and is the simulator that has been used in the automata course at MDH among others. It has a wide range of different automaton types to simulate and a lot of features that help the user to manipulate automata without using pen and paper.

Features

- Supports FA, PDA, TM, Grammar, RE and more
- Save image to BMP or JPEG and more
- Convert NFA ↔ DFA
- Convert RE ↔ NFA
- Convert Grammar ↔ NFA
- Determination and Minimization

Review

You select your preferred language in the starting menu. By start creating FA you simply select Finite Automata in the menu. After selecting automata type you get the editor page, with a toolbar and a workspace to create your automata.

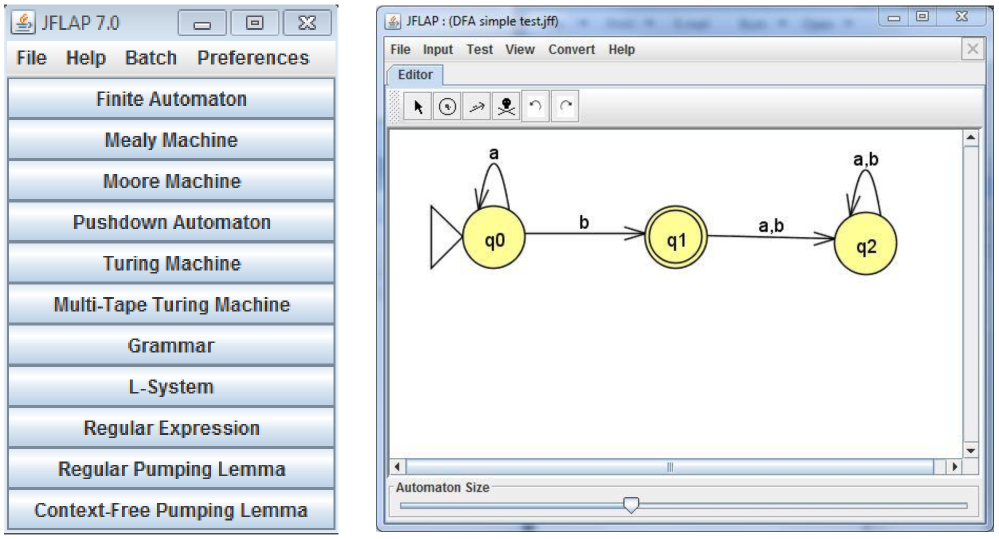
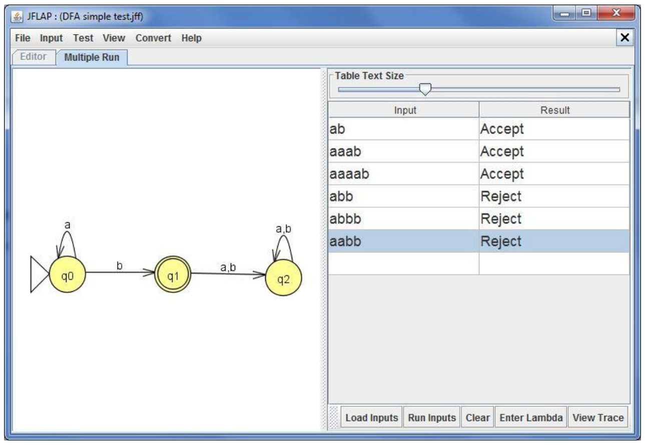


Figure 13 Selection menu and editor view

By selecting the “state” button you can create new states. The names on the states are created automatically and by right clicking you can set if initial and final. Next to the state tool is the ability to create transitions. Transitions can be dragged from one state to the next state or looped.

By pressing the scroll button on your mouse you can also curve the transitions to make your automata look better. By using any options under “input” you can test run your automata. There are four options. Test the input with Closure, Step by State, Fast Run and Multiple Run.
The “Multiple run” option lets you test multiple inputs with either a reject or an accepted result.



Input	Result
ab	Accept
aaab	Accept
aaaab	Accept
abb	Reject
abbb	Reject
aabb	Reject

*Figure 14 JFLAP with the language a*a with multiple debug.*

Creating DFA and NFA (both combined as FA in JFLAP) is fast and you do not have to set any variables or labels by adding items. Users do not have to set any alphabet before debugging which makes this simulator very straightforward. By double left clicking any transition the user can easily setup every transition.

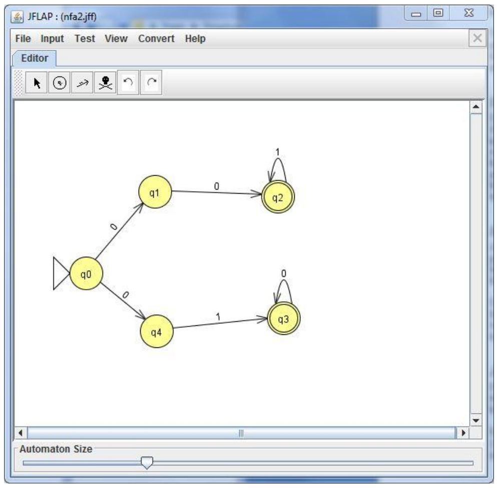


Figure 15 JFLAP with NFA representing the language (001)|(010*)*

FA, PDA and Turing machines are very similar to create and edit with some differences like transitions and which algorithms (like automata conversion) you can apply.

Conclusion: JFLAP

Functionality

- JFLAP have enough features for the FABER course and more.
- Determination and Minimization for automata is available.
- Images can be made from current automata easily.

Tools

It is easy to debug your creations with one or multiple inputs. JFLAP responds well and changes active states to grey when the “Step by State” feature is selected. There should have been some interactive animation for each transition which JFLAP does not have. Typing in a single input is possible where the user either gets a rejected or accepted state when done.

User Friendliness

It’s easy to create Finite Automata, you simply point and click every state and drag transitions between them. Names are added automatically each state. Same with PDA’s which are created just like FA’s. Turing machines do use the same interface and look. It is easy for beginners to just create an automaton with minor knowledge. Some features require some modifications on your automata to work properly which is often noted on the home page. For example PDA to Grammar conversion requires the PDA to follow specific rules before it will work.

Design

JFLAP is well structured and have a simplistic layout. Creating states and transitions are simple and straightforward. The start menu separates all the different types of automata and languages and selecting one of them gets you to the editor screen which is mostly the same with minor differences. The debugging could need some moving animations which it is not available in JFLAP.

Documentation

The JFLAP home page has a very well written guide which is very helpful if you get stuck.

Compatibility

To use JFLAP JavaSE1.4 is needed. It is easy to run the .jar file that contains the program on any platform that uses Java. JFLAP is the best known automata simulators out there and it covers most of the needed material found in an automata course.

Simulator: PetC

Home Page: http://people.dsv.su.se/~henrikbe/petc/
Current Version: Version 0.11

Description

This is a FSA simulator by Henrik Bergström, a teacher at the University of Stockholm [9].

Features

- Supports DFA and NFA
- Unfortunately, as the creator clearly states on the homepage, the application is far from complete.

Review

You can have multiple automatons in the same editor window. Here you can create new automata or open an existing one. You can simply add new states and transitions. States automatically gets their name when they are created. You need to specify your alphabet set before you continue. The A button in the toolbar opens a menu where you can specify the Alphabet.

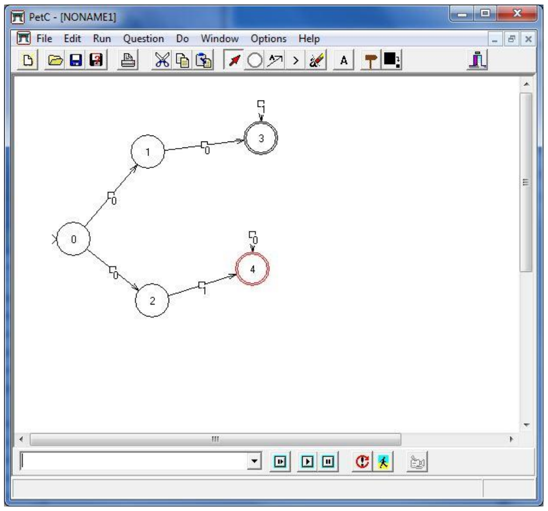


Figure 16 PetC with the language (001)|(010*)*

Below the workspace is a toolbar where you can debug your automaton. You simply put in the input string in the textbox and press “Run”. You will see that the program eats the string until a message box pops up with either an accepted or rejected state. It works well to create both DFA and NFA.

Conclusion: PetC

Functionality

With PetC the user can edit more than one automaton. PetC is missing any conversion feature and both Determination and Minimization.

Tools

The debug functions can be found beneath the workspace where you edit the automata. The debugging is animated with the current state either colored in red or green.

User Friendliness

The editor works as expected it’s easy to create more simple automatons.
Setting the alphabet could be simpler. Now you must click on the letters you want in a table.

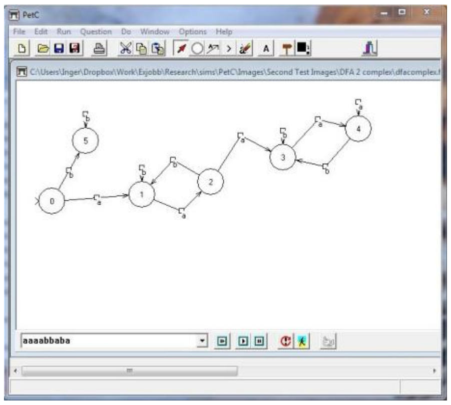


*Figure 17 PetC with the language L = aWaaWa, W = {a,b}**

Design

States and transitions is not rendered at all while you drag them around, having the states moving alongside the mouse while dragging feels more responsive.

Documentation

There is a tutorial available on the home webpage that goes through the basic parts of the application.

Compatibility

This program was currently only available for download as a Windows binary which is not good for non-Windows users.

Simulator: Tuatara Turing Machine Simulator

Home Page: http://tuataratmsim.sourceforge.net/
Current Version: 1.0

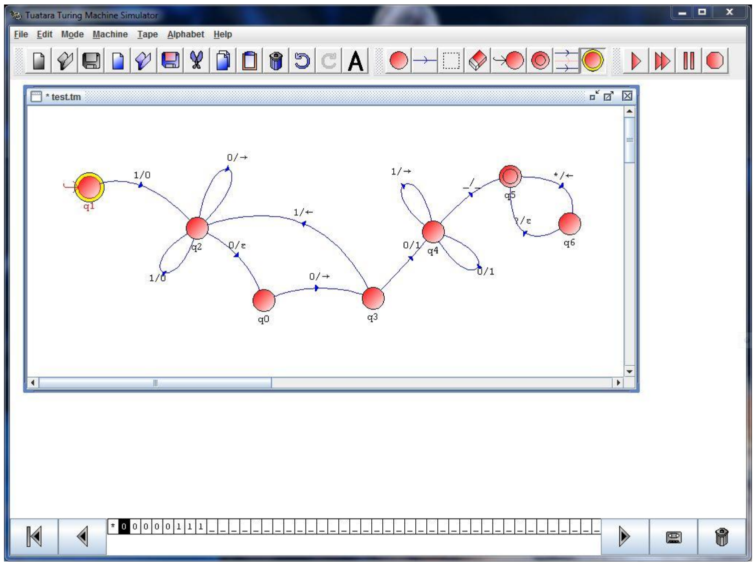
Description

Tuatara Turing Machine Simulator is written by Jimmy Foulds with founding from the Department of Mathematics at the University of Waikato. [7]

Features

- Supports Turing machines
- Uses several Turing machines simultaneously

Review



Tuatara Turing Machine Simulator
File Edit Mode Machine Tape Alphabet Help
* test.tm
q1
q2
q0
q3
q4
q5
q6
1/0
0/→
1/←
0/ε
1/0
0/→
1/→
0/1
0/1
_/
*/←
?/ε

Figure 18 TTMS with Turing machine

States and Transitions can easily be added to the workspace. The user can create new automata in the same window. Before running your Turing machine, both initial, final state and the alphabet must be set. Below is a tape the user can type in while the Turing machine is running. During debugging the current state is marked with a yellow color. The whole machine can be executed at once or the user can step states using the play button. Each transition is set by pressing any letter on your keyboard. There are two variables in each transition which is an input value and one action. Epsilon is chosen by pressing *shift+e*. moving the head is done with the direction keys.

Conclusion: Tuatara Turing Machine Simulator

Functionality

- This simulator only handles Turing machines
- There is no way of converting TM to Grammar or regular expressions.

Tools

Creating Turing machines is simple and fast, but transitions are specified differently which is confusing. There is an input and an action or write to set at each transition which makes creation of Turing machines more confusing. Debugging is easy, you select the start point and reset the tape then you can run your program. When executing or stepping the Turing machine each stage is colored and the tape is updated until the final stage is reached. If something went wrong you are noticed with a warning message, for example if the writer gets behind the starting point of the tape.

User Friendliness

The editor feels good to work with but there were few problems during use, for example Tuatara could be more helpful explaining how to create transitions.

Design

The layout is similar to what you find in most simulators. You can have many Turing machines in the same window and with one tape for each instance.

Documentation

There is little information about the program on the home webpage but do not come with any manual.

Compatibility

Tuatara comes in plain Java source code that you have to compile. I only got it working well using NetBeans 7.0.1 using Windows 7 after a while. Getting started with the source code could be troublesome for some.

Using this simulator with Turing machines related assignments would work but they don't follow the same rules as found in the course book or the lectures.

Simulator: Turing Machine Simulator

Home Page: http://ironphoenix.org/tril/tm/
Current Version: 1.0

Description

A web based simulator for Turing machines. It is created by Suzanne Britton which has the applet on her own website [3].

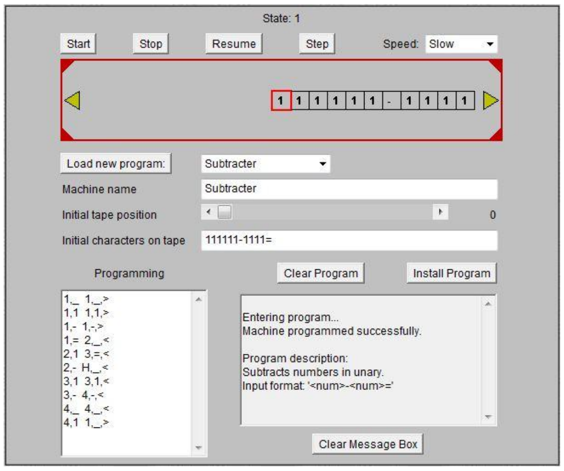
Features

As the name says it does Turing machines where you program your own Turing machine. The user can step and pause during the execution of the active program and the tape is updated every step. Programs can be saved and loaded into the simulator.

Review

Load New Program loads a created program which is the Turing machine (see Figure 19). The box under *Programming* is the currently loaded program which can be run by pressing *Start* in the top corner. The speed of the execution and *Stepping* can be set with the buttons at the top of the window. The red box is the tape. The console right below shows useful messages about the program. Turing machines are created using the programming box. Every line uses the following this structure.

[state],[character],[new state],[new character],[direction]



Load new program:	Subtracter
Machine name	Subtracter
Initial tape position	0
Initial characters on tape	111111-1111=

Programming

```
1,_  1,_,>
1,1  1,1,>
1,-  1,-,>
1,=  2,_,<
2,1  3,=,<
2,-  H,_,<
3,1  3,1,<
3,-  4,-,<
4,_  4,_,<
4,1  1,_,>
```

```
Entering program...
Machine programmed successfully.

Program description:
Subtracts numbers in unary.
Input format: '<num>-<num>='
```

Figure 19 Turing Machine Simulator

Creating Turing machines with Turing Machine Simulator from scratch is more complicated than using a graphical editor.

Conclusion: Turing Machine Simulator

Functionality

The user can create Turing machines using a line by line programming box.

Tools

When running programs the tape is animated well and it is easy to see where on the tape you are. You can easily move around with two arrows on the sides of the tape. Obviously you do not have a view of how the Turing machine looks like.

User Friendliness

You need to know how to program a Turing machine in Turing Machine Simulator which is time consuming. There are a couple of premade Turing machines to study by loading them and press start to see the TM running with the characters on the tape. As said the program is simple but it is a bit harder to learn and use on your own.

Design

The layout is dated but yet very simple.

Documentation

There is a guide on the webpage that explains the syntax and how to use the application.

Compatibility

The simulator is available as source code and as an applet on the home webpage with instructions. With an assignment with Turing machines this program can work well.

Simulator: Visual Automata Simulator

Home Page: http://www.cs.usfca.edu/~jbovet/vas.html
Current Version: Version 1.2.2

Description

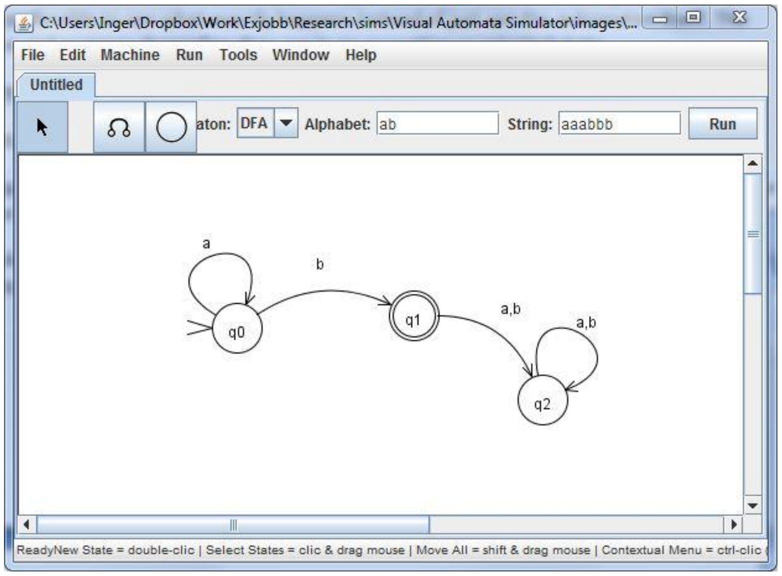
This application is created by Jean Bovet from the University of San Francisco.
According to the home website this simulator is created with the inspiration from an automata and object-orientation course [1].

Features

- Supports DFA, NFA, Turing Machines
- Convert NFA → DFA
- Save to BMP, JPEG and more.

Review

When the application starts you begin directly in the editor (see Figure 20). Either the user can use a DFA or NFA or by pressing *new* to change to a Turing machine. The toolbar is simple and contains a select, state or transition buttons. The user can change the automaton type between DFA and NFA. The alphabet can be set in the Alphabet textbox and a string can be set as input. The user simply press run to test that string.

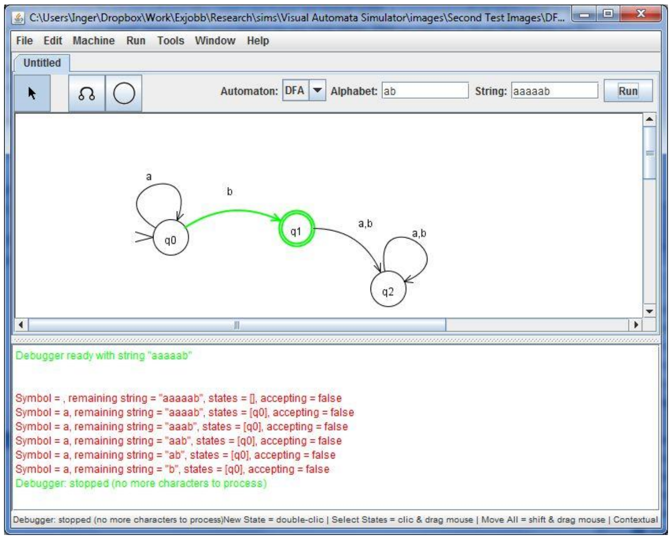

C:\Users\Inger\Dropbox\Work\Exjobb\Research\sims\Visual Automata Simulator\images\...
File
Edit
Machine
Run
Tools
Window
Help
Untitled
aton: DFA
Run
a
b
a,b
a,b
q0
q1
q2

Alphabet: ab
String: aaabbb

ReadyNew State = double-clic | Select States = clic & drag mouse | Move All = shift & drag mouse | Contextual Menu = ctrl-clic

*Figure 20 Visual Automata Simulator with the language a*b*

By selecting Run then Debug enables you to debug the automata step by step. The automaton gets animated where the active transition and state gets red or green depending if the state is final. Below is also a console like output where the user can inspect every step.


C:\Users\Inger\Dropbox\Work\Exjobb\Research\sims\Visual Automata Simulator\images\Second Test Images\DF...
File Edit Machine Run Tools Window Help
Untitled
Automaton: DFA
Alphabet: ab
String: aaaaab
Run
a
b
q0
q1
a,b
a,b
q2

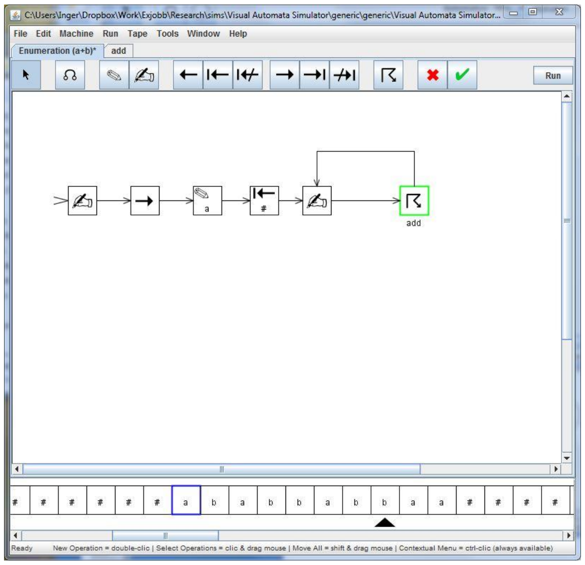
```
Debugger ready with string "aaaaab"


Symbol = , remaining string = "aaaaab", states = [], accepting = false
Symbol = a, remaining string = "aaaab", states = [q0], accepting = false
Symbol = a, remaining string = "aaab", states = [q0], accepting = false
Symbol = a, remaining string = "aab", states = [q0], accepting = false
Symbol = a, remaining string = "ab", states = [q0], accepting = false
Symbol = a, remaining string = "b", states = [q0], accepting = false
Debugger: stopped (no more characters to process)
```

Debugger: stopped (no more characters to process)New State = double-clic | Select States = clic & drag mouse | Move All = shift & drag mouse | Contextual

Figure 21 Debugging in Visual Automata Simulator

There simulator can handle Turing machines. However the notation is difficult to understand. Here every TM state is split down into several functions which the user needs to learn how to use them properly.


C:\Users\Inger\Dropbox\Work\Exjobb\Research\sims\Visual Automata Simulator\generic\generic\Visual Automata Simulator...
File Edit Machine Run Tape Tools Window Help
Enumeration (a+b)*
add
Run
a
#
add
# a b a b b a b b a a # # #

Ready New Operation = double-clic | Select Operations = clic & drag mouse | Move All = shift & drag mouse | Contextual Menu = ctrl-clic (always available)

Figure 22 Visual Automata Simulator with Turing machine

Conclusion: Visual Automata Simulator

Functionality

Converting NFA to DFA is one of the only implemented features. For example determination and minimization is not available. Pushdown Automata is also not available.

Tools

The debugging feature works well. The user can type an input with a corresponding alphabet and by pressing the “Debug one step” makes the simulator to go one step. The problem is that you have to go every time to the upper menu to select the option to go one step. I think that having a debug output somewhere is preferable to see what exactly happens instead of just only the automata. The states either are red or green depending they are in an accepting or rejected state.

User Friendliness

It is easy to create finite automata. The Turing machine part of the editor is not similar and is harder to learn on your own. There are some misconceptions like Lambda transitions that are not automatically created when adding a blank transition. This result in a rejected state although I thought that it should be accepted. Luckily lambda or epsilon is set as “e” in the settings menu.

Design

The layout is nice and simple and is very similar to JFLAP. States and transitions can be moved around nicely.

Documentation

The home webpage is simple with some images and videos but no mayor documentation or guides. Like the lambda transitions it makes it hard to know if the functionality is there or not.

Compatibility

Visual Automata Simulator uses Java and should like JFLAP run well on most platforms.
While testing the application using Windows 7 the application could freeze without any error message.
Using this simulator in course assignments has potential with FA but is too complicated for assignments with Turing machines.

3.2 Simulator Analysis Discussion

All of the collected simulators have been tested. To make it easier the results is collected for easier overview. Most programs had problems with sparse features and documentation which creates limitations that is harder to use. JFLAP is the only simulator with a full range of features (see Table 3). Most of the programs can at least edit and create two types of automata, but very few of them that can convert Regular Expressions or Grammar to a Finite Automata. JFLAP can also handle TM which most simulators that aim at DFA and NFA do not have yet.

Sim	DFA	NFA	PDA	TM	RE	GM	RE↔FA	GM↔FA
Automata Editor	x	x	-	-	-	-	-	-
Auto Ed by Max	x	x	-	-	-	-	p	-
Automaton Sim	x	x	x	x	-	-	-	-
JFAST	x	x	x	x	-	-	-	-
JFLAP	x	x	x	x	x	x	x	x
PetC	x	x	-	-	-	-	-	-
Tuatara	-	-	-	x	-	-	-	-
TMS	-	-	-	x	-	-	-	-
VAS	x	x	-	x	-	-	-	-

x	Available
p	Partially available
-	Not available

Table 3 Automata types available

All tests where easy to create in JFLAP. Automata Editor, Automaton Simulator and Visual Automata Simulator work well to edit automata. Simulators like JFAST did not work well because of the number of bugs that where found during the tests (see Table 4 and 5).

Simulator	Fun	Tools	User friendliness	Design	Com	Doc	Tot
Automata Editor	2	4	3	3	3	1	16
Auto Ed by Max	2	3	1	2	3	1	12
Automaton Sim	2	4	3	3	5	3	20
JFAST	3	1	1	2	5	1	13
JFLAP	5	4	4	3	5	5	26
PetC	2	3	2	2	1	4	14
Tuatara	3	3	1	4	2	3	16
TMS	2	1	1	1	5	4	14
VAS	2	3	3	4	5	3	20

Grade from 1 to 5 where 1 is the worst and 5 is the best

Table 4 Grades for each simulator

JFLAP is still in development with a new version coming soon with more improvements. There is a higher possibility that improvements will be released more regularly. JFLAP have better documentation which most programs lacks. Important is that JFLAP is very simple and users can use the software with ease just by using it.

Sim	DFA	NFA	PDA	TM	Tot
Automata Editor	3	2	0	0	5
Auto Ed by Max	1	1	0	0	2
Automaton Sim	3	3	2	2	10
JFAST	2	2	2	1	7
JFLAP	3	3	3	3	12
PetC	3	3	0	0	6
Tuatara	0	0	0	1	1
TMS	0	0	0	1	1
VAS	3	3	0	1	7

3	All tests successful
2	Tests partially successful
1	Tests unsuccessful
0	Not implemented

Table 5 Test cases for each automata type

JFLAP do have all the best perks and is the best all-round choice in comparison of the other simulators. There is no reason to pick any other program because they have not enough documentation or features. Using JFLAP for all the course work will be both more efficient and helping students than slowing them down by roughly implemented software.

4. CONCLUSIONS ON THE CHOICE OF SIMULATOR

JFLAP is the definitive choice when choosing automata simulators in a desktop environment. Other candidates either have fewer features or did not perform any better than JFLAP. Tests were obvious where simple tasks that JFLAP did very well were tedious in other programs. The problem with comparison between automata simulators from the internet is that their functionalities are different and often difficult to match with each other. Nevertheless, even though some particular functions in some simulators may be better, in general JFLAP functionality is superior.

Given JFLAP as a simulator of choice, the manual and assignments are written for use with JFLAP. From the manual and assignments the user can learn how to use the program and how to create basic automata and languages.

Writing this report was useful to learn how to create automata for assignments.
Also there was testing of software which was a good exercise in studying software and deciding which are best.

5. FUTURE WORK

There is a lot to improve in automata simulation, as in most simulator programs tested the interface is often outdated. New programs that make such improvements are a good idea to review and consider using in the future. Computers and phones with touch screen control are today more common than ever. Automata simulation using touch control would be interesting where students and teachers can learn about automata in a more interactive and entertaining way. Testing simulators for popular mobile platforms is a good idea and could be an alternative to JFLAP. Currently I have not seen anyone making an automata simulator for mobile platforms. To improve exercises for the FABER course it is a good idea to create and maintain our own automata simulator that fit our own needs. With software we understand we can quickly add the features we need.

6. REFERENCES

[1] Visual Automata Simulator, http://www.cs.usfca.edu/~jbovet/vas.html (16-10-12)

[2] jFAST finite automata simulator, http://sourceforge.net/projects/jfast-fsm-sim/ (4-1-13)

[3] Turing Machine simulator, http://ironphoenix.org/tril/tm/ (4-1-13)

[4] JFLAP, http://www.jflap.org/ (4-1-13)

[5] Automata Editor, http://automataeditor.sourceforge.net/ (4-1-13)

[6] Automaton Simulator, http://ozark.hendrix.edu/~burch/proj/autosim/ (4-1-13)

[7] Tuatara Turing Machine Simulator, http://tuataratmsim.sourceforge.net/ (4-1-13)

[8] Automata Editor, http://max99x.com/school/automata-editor (4-1-13)

[9] PetC, http://people.dsv.su.se/~henrikbe/petc/ (4-1-13)

[10] Mississippi State University course: Introduction to Formal Languages and Automata, http://www.cse.msstate.edu/~cse3813/summer10/ (4-1-13)

[11] P. Linz, An introduction to FORMAL LANGUAGES AND AUTOMATA 5th ed, Jones And Barlett

[12] P. Linz, S. H. Rodger, JFLAP Activities for Formal Languages and Automata, included CD-ROM from. [11]

[13] J. J. Cogliati, F. W. Goosey, M. T. Grinder, B. A Pascoe, R. J. Ross, C. J. Williams, *Realizing the Promise of Visualization in the Theory of Computing*, Montana State University, Montana Tech of the University of Montana

[14] S. Reading, J. Lim, S. H. Rodger, *Increasing Interaction and Support in the Formal Languages and Automata Theory Course*, Computer Science Department, Duke University

[15] R. Reis, N. Moreira, A. Almeida, *GUItar and FAgoo: Graphical interface for automata visualization*, editing, and interaction, DCC-FC & LIACC, Universidade do Porto

[16] GRAPHVIZ, open source graph visualization software, http://www.graphviz.org/ (4-1-13)

[17] W. Yurcik, M. L. Cobo, C I. Chesñevar, *Using Theoretical Computer Simulators for Formal Languages and Automata Theory*, Department of C.S. and Engineering, Univ. Nacional del Sur

7. APPENDICES

The following appendices are found in section 8: MANUAL AND EXERCISES is explained in this section.

7.1 Appendix 1

Even though JFLAP is provided with a manual, we produced a condensed version of the manual suitable for classroom use. Explaining all features would make the guide too long which is unnecessary. For example features like the Mealy and Moore machines will not be discussed. A student that have done work with FA and also have read this guide should not have any problems using other features of JFLAP as well.

7.2 Appendix 2

The assignments are based on the use of JFLAP. Assignments introduce the student to JFLAP and its features by creating different types of automata.

Assignment 1

The first assignment goes through FA with DFA and NFA, Grammar and Regular Expressions.

Assignment 2

The second assignment handles Pushdown Automata and Context Free Grammar.

Assignment 3

The third assignment handles Turing machines and restriction free languages.

8. MANUAL AND EXERCISES

8.1 Appendix 1. Manual for JFLAP simulator use in the Formal Languages, Automata and Theory of Computation course

JFLAP User Manual

For JFLAP version 7.0

Content

INTRODUCTION 2

WHAT IS JFLAP? 2
SETUP 2
USING JFLAP 2
ADDITIONAL RESOURCES 2

FINITE AUTOMATA AND REGULAR LANGUAGES 3

FINITE AUTOMATA 3
SHOW NONDETERMINISM 8
CONVERT NFA TO DFA 9
REGULAR EXPRESSIONS 10
CONVERT FA TO REGULAR EXPRESSION 11
CONVERT FA TO GRAMMAR 12

CONTEXT FREE GRAMMAR AND PUSHDOWN AUTOMATA 14

PUSHDOWN AUTOMATA 14
GRAMMAR 15
CONVERT PUSHDOWN AUTOMATA TO GRAMMAR 15

RESTRICTION FREE LANGUAGES AND TURING MACHINES 17

TURING MACHINE 17

Introduction

What is JFLAP?

JFLAP is a program that makes it possible to create and simulate automata. Learning about automata using pen and paper can be both difficult and time consuming. With JFLAP on the other hand you can create automata of a variety of different types and it's easy to change your creation whenever you want. JFLAP supports creation of DFA and NFA, PDA, Turing Machines, Grammar, Regular Expressions and more.

Setup

JFLAP is available from the homepage: (www.JFLAP.org). From there press "Get FLAP" and follow the instructions. You will notice that JFLAP have a .JAR extension. This means that you need Java to run JFLAP. With Java correctly installed you can simply select the program to run it. You can also use a command console run it from the files current directory with, *Java –jar JFLAP.jar.*

Using JFLAP

When you first start JFLAP you will see a small menu with a selection of eleven different automata and rule sets. Choosing one of them will open the editor where you create chosen type of automata. Usually you can create automata containing states and transitions but there is also creation of Grammar and Regular Expression which is made with a text editor.

Additional Resources

Recommended Reading: JFLAP - An Interactive Formal Languages and Automata Package
Rodger, Finley, ISBN: 0763738344

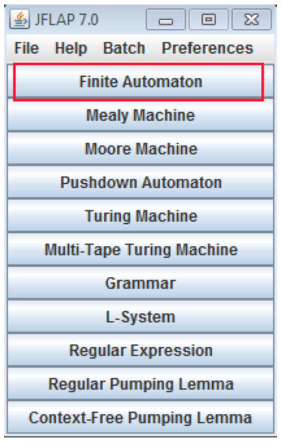
JFLAP assignments for JFLAP - An Interactive Formal Languages and Automata Package
http://www.cs.duke.edu/csed/jflap/jflapbook/files/

Getting Started With JFLAP, Colorado State University
http://www.cs.colostate.edu/~massey/Teaching/cs301/RestrictedAccess/JFLAP/
gettingstarted.html

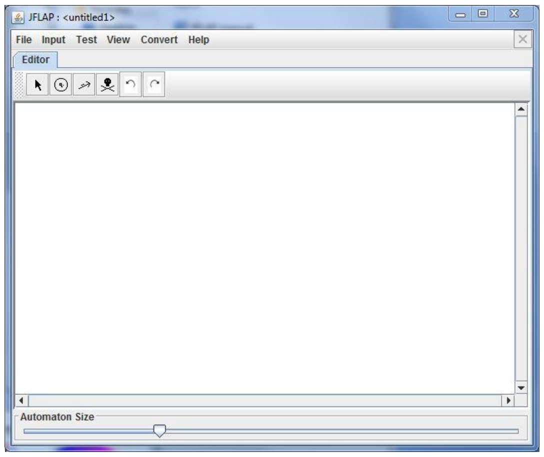
Finite Automata and Regular Languages

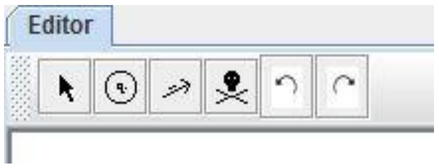
Finite Automata

The simplest type to begin with is Finite Automata which is the first option from the selection menu. In JFLAP both DFA and NFA is created using Finite Automata.



Now you should have an empty window in front of you. You will have a couple of tools and features at your disposal.

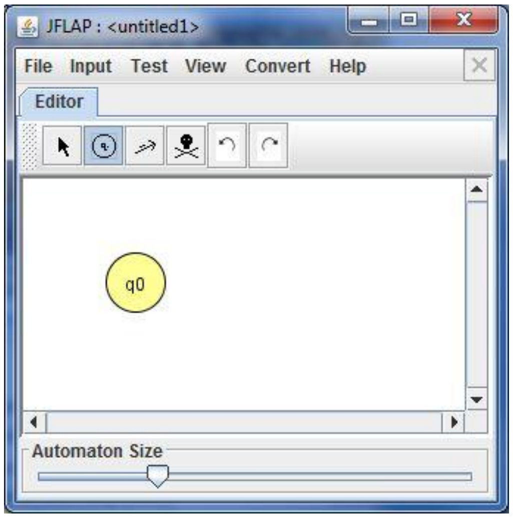




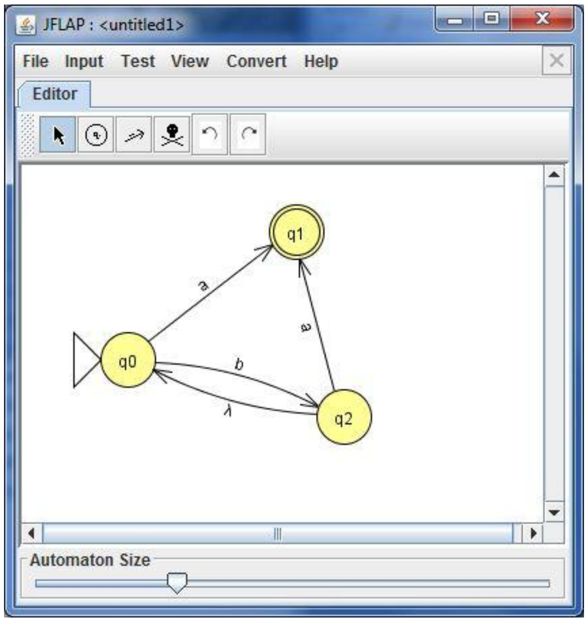
The toolbar contains six tools which are used to edit automata.

Attribute Editor Tool, change properties and position of existing states and transitions.
State Creator Tool, creates new states.
Transition Creator Tool, creates transitions.
Deletion Tool, deletes states and transitions.
Undo/Redo, changes the selected object prior to their history.

Creating an automaton is easy with the state and transition tools. Note that you need to change back to the Attribute Editor Tool (first) to change states. Let’s try to add states with the State Creator Tool (second).

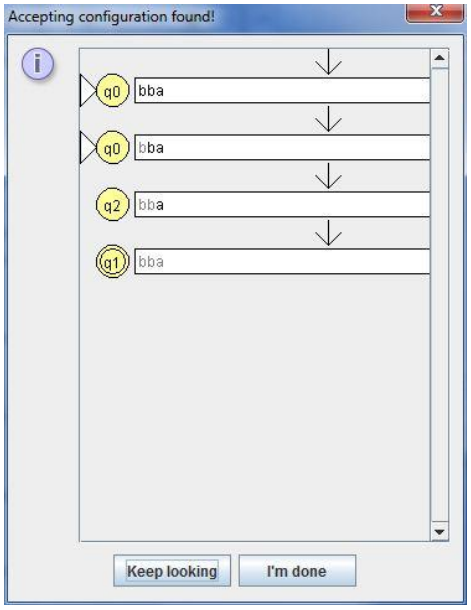


When adding states they automatically get a name assigned to them which can be changed using the Attribute Editor Tool. Transitions are easily dragged between states with the mouse using the Transition Creator Tool.

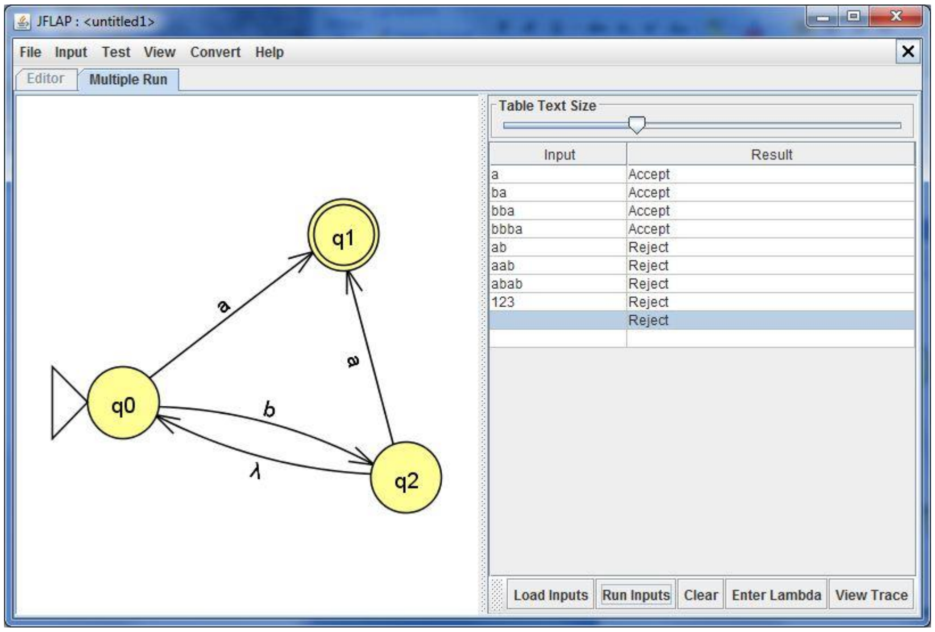


This automaton only accepts strings containing b but end with a. To test this automaton you can use any of the available tools under the Input menu.

The fastest in this case is to use Input→Fast Run. A menu where you can set your input string pops up. Type the string “bba” and select “OK”. The program shows all the transitions that are done when consuming the input string.

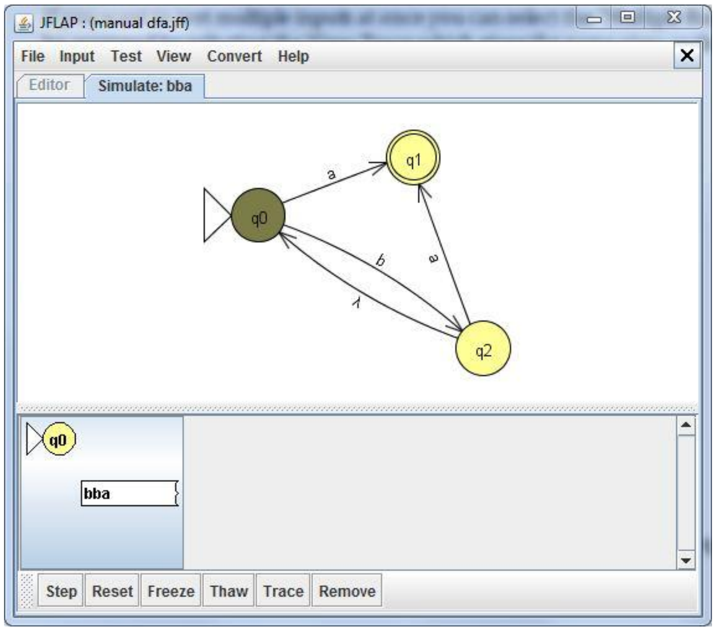


If you want to test multiple inputs at once you can select the “Multiple Run” option. If you wish to individually review single runs can be accessed by selecting the View Trace, which gives a view similar to the “Fast Run” option.

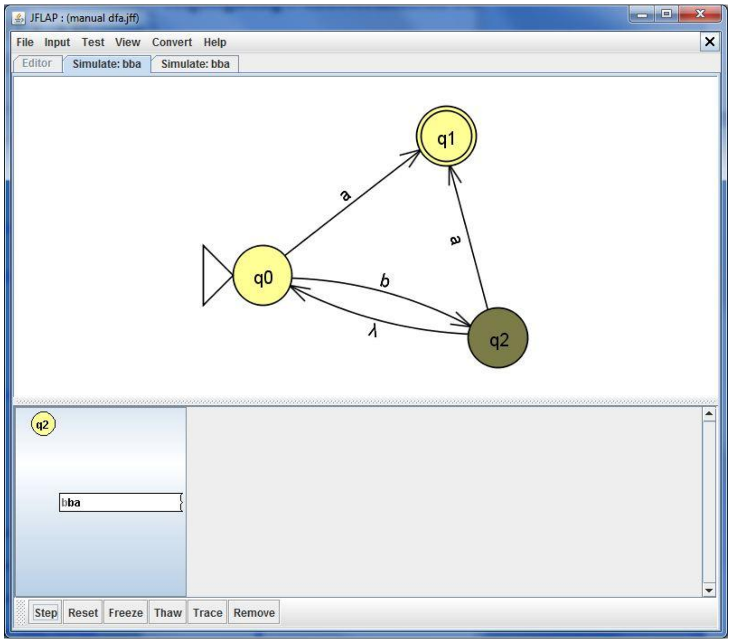


Input	Result
a	Accept
ba	Accept
bba	Accept
bbba	Accept
ab	Reject
aab	Reject
abab	Reject
123	Reject
	Reject

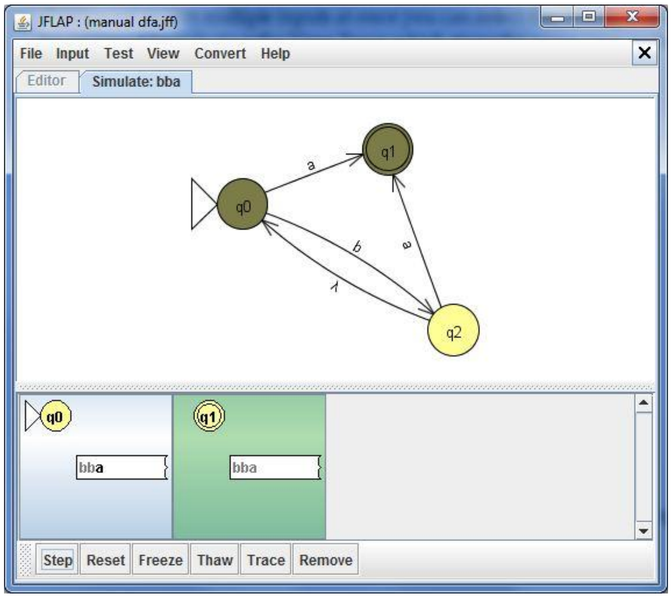
To debug the automata itself “Step by Closure” or “Step by State..” can be selected.
For each step the automata highlights the currently active state. By “stepping” you press the “Step” button.


JFLAP : (manual dfa.jff)
File Input Test View Convert Help
Editor
Simulate: bba
q0
q1
q2
a
b
a
λ
q0
bba
Step Reset Freeze Thaw Trace Remove

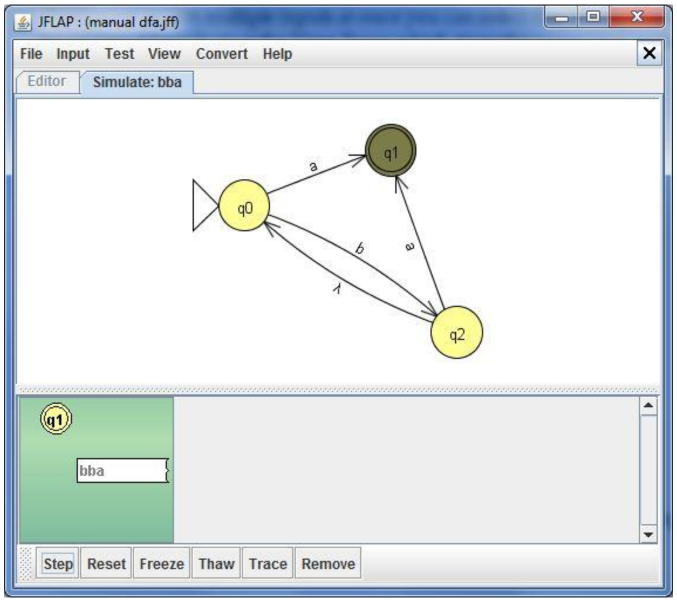
The first character consumed. Notice that the first letter is grayed out and the currently active state have changed.



Now the last character is about to be consumed. This step shows the transition between q0 and q1. If there happen to be multiple paths with same character you will see them grayed out.


JFLAP : (manual dfa.jff)
File Input Test View Convert Help
Editor
Simulate: bba
a
q0
q1
b
a
λ
q2
q0
bba
q1
bba
Step Reset Freeze Thaw Trace Remove

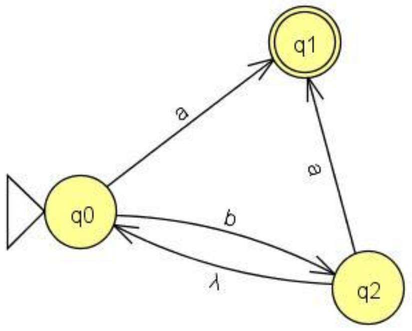
The simulator reached the final state.


JFLAP : (manual dfa.jff)
File Input Test View Convert Help
Editor
Simulate: bba
a
q0
q1
b
a
λ
q2
q1
bba
Step Reset Freeze Thaw Trace Remove

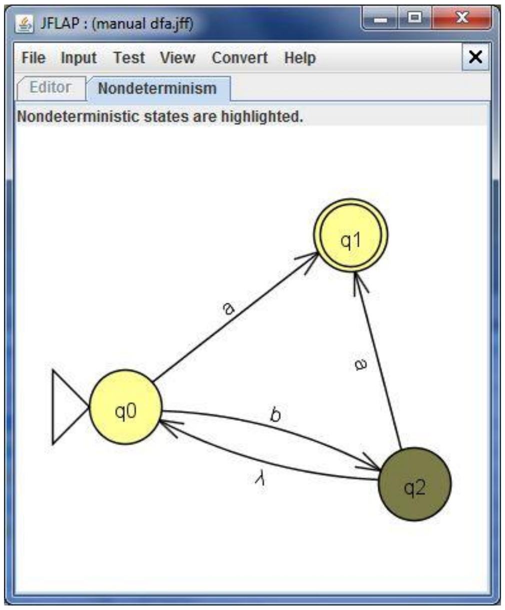
To restart the test you can select the reset button to start from the beginning or press the X button in the top right corner to go back to the editor.

Show Nondeterminism

Creating NFA is no different to creating DFA. Finite Automata is considered either a DFA or NFA depending if there is lambda transitions or paths with same symbols. The previous automaton is a NFA because of the lambda transition where you can access all paths from the q2 state.



It is possible to let JFLAP determine if the automaton is a non-deterministic automaton. Select Test then “Highlight Non-Determinism”.

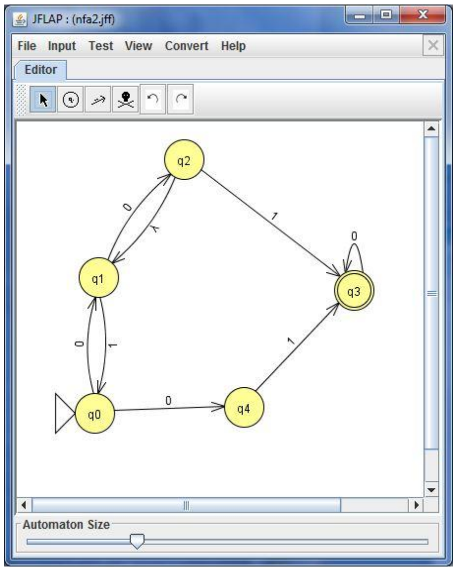


Nondeterministic states are highlighted.

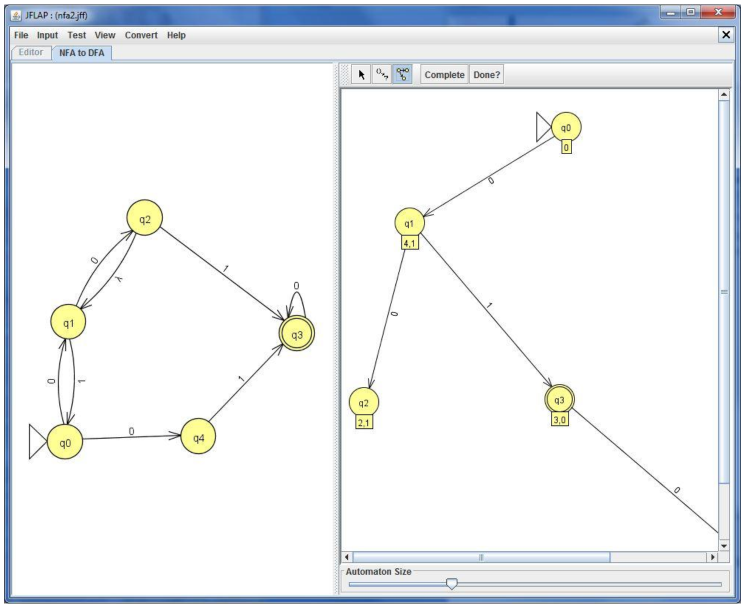
This simply shows that the q2 state is non-deterministic.

Convert NFA to DFA

JFLAP can convert NFA to DFA. This automaton has two nondeterministic states which could easily be changed back to a DFA.

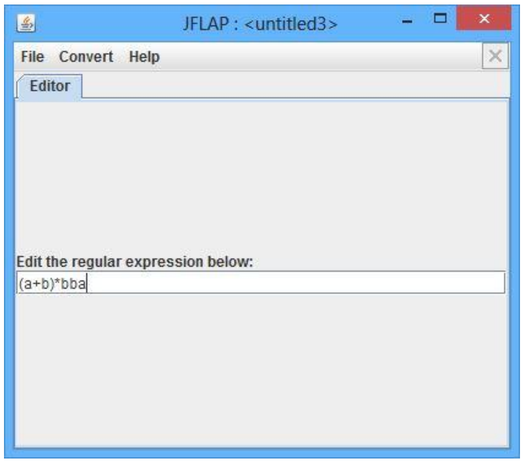


Create a NFA and then choose Convert→Convert to DFA. This will open the conversion view where you either let JFLAP do the work or try yourself to convert it. The left view is the original automata and the right one is the new DFA. Use the state expander tool to expand the states until the DFA is complete. Using the *Complete* button will automatically create the whole DFA for you. The *Done?* button will tell if the DFA is either done or not. Once the DFA is complete it will be exported to a new JFLAP window with your converted DFA.



Regular Expressions

Regular Expressions can be typed into JFLAP which can then be converted to an NFA.



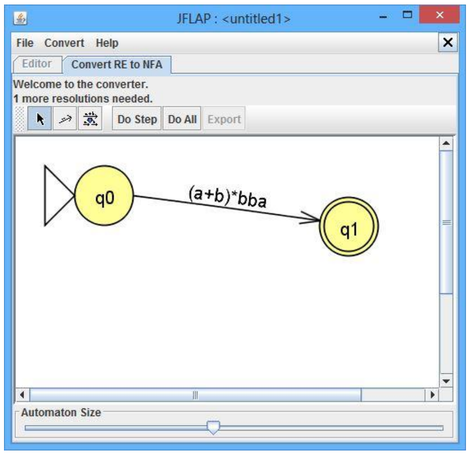
Edit the regular expression below:

(a+b)*bba

Choose Regular Expression in the main menu then just type the expression in the textbox.
Definitions for Regular Expressions in JFLAP:

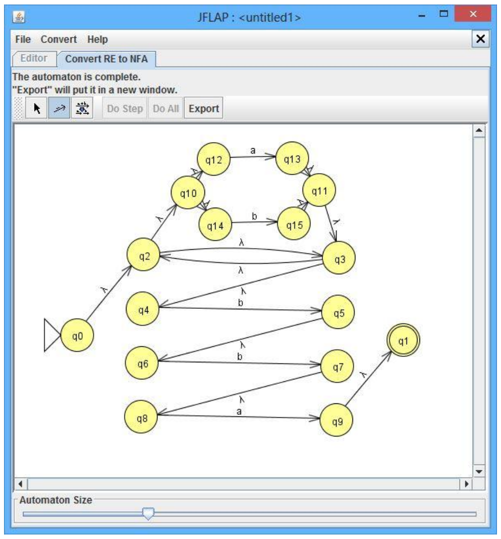
- * Kleene Star
- + Union
- ! Empty String

Correctly written expressions can then be converted to a NFA. To convert your expression select Convert→Convert to NFA. The conversion will begin with two states and a transition with your Regular Expression. With the *(D)e-expressionify Transition* tool you can break down the Regular Expression into smaller parts. Each transition will contain a sub expression. The next step is to link every rule with lambda transitions. Add new transition between states that should be connected with the Transition Tool. If you are unsure what to do you can select *Do Step* to automatically make the next step. If you want the NFA immediately *Do All* creates the whole NFA for you.



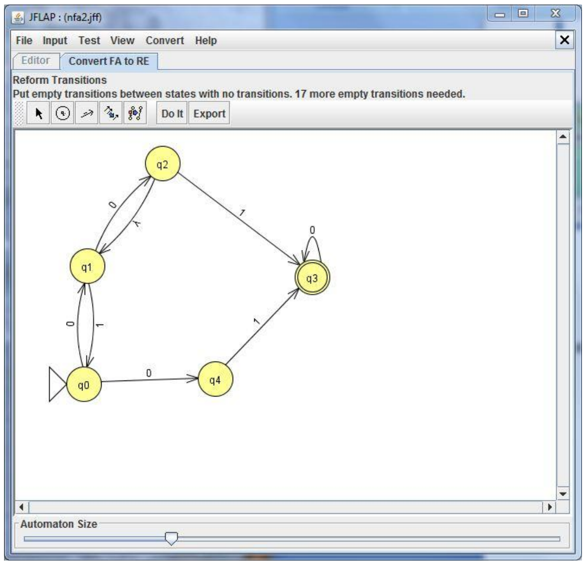
Welcome to the converter.
1 more resolutions needed.

You can notice how the conversion differs depending on how the Regular Expression looks. For example the expression *a+b* results in a fork, were either ‘a’ or ‘b’ can be chosen.



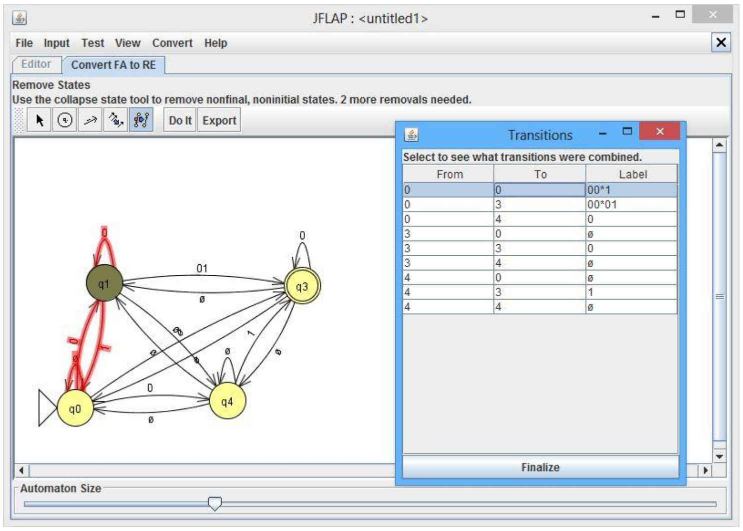
The automaton is complete.
"Export" will put it in a new window.

Convert FA to Regular Expression



Reform Transitions
Put empty transitions between states with no transitions. 17 more empty transitions needed.

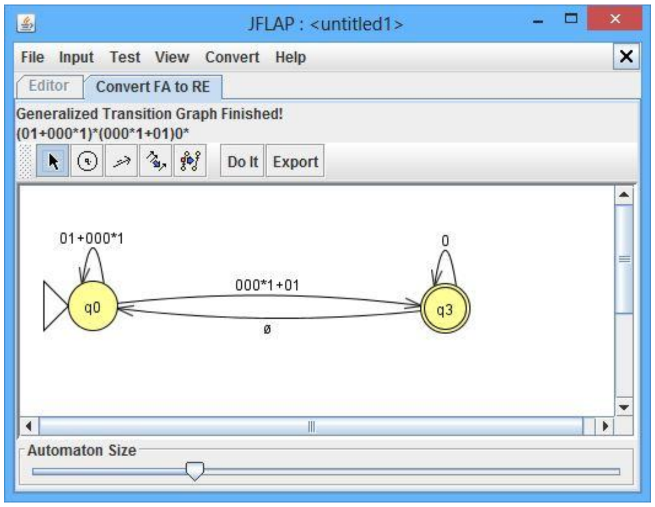
Follow the instructions above the toolbar. To make the conversion work, empty transitions must be added between states that have yet no transition. States that either is Initial or Final must be removed which you do with the collapse state tool. With the collapse tool you can use the table to inspect combined transitions from that state. The state is removed with The Finalize button.


Remove States
Use the collapse state tool to remove nonfinal, noninitial states. 2 more removals needed.

Select to see what transitions were combined.

From	To	Label
0	0	00*1
0	3	00*01
0	4	0
3	0	ø
3	3	0
3	4	ø
4	0	ø
4	3	1
4	4	ø

When all the necessary steps are made, the converted automata contain the regular expression. You can also see the complete regular expression above the toolbar which can be exported using *Export*.

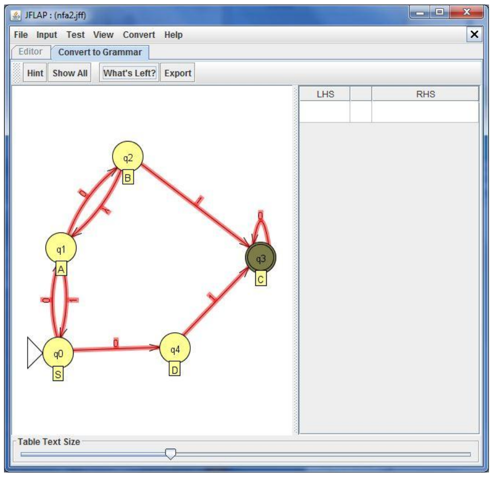

Generalized Transition Graph Finished!
(01+000*1)*(000*1+01)0*

JFLAP is capable to convert the regular expression to a NFA again. If the original automata is a DFA the result might differ because JFLAP add a lot of lambda transitions. You might need to convert further to a minimized DFA to get your automata back.

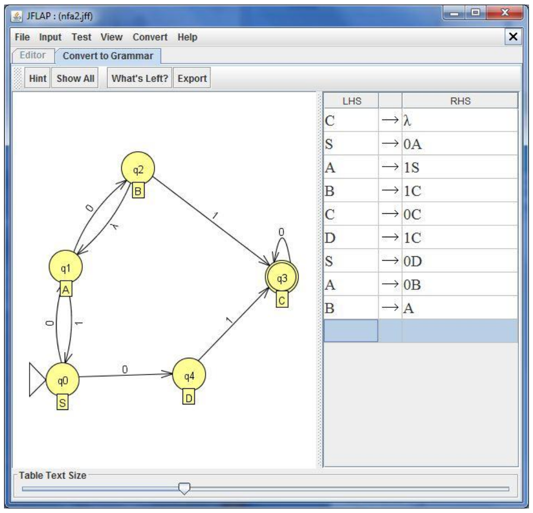
Convert FA to Grammar

When using a Finite Automaton select Convert→Convert to Grammar. The conversion view will contain your automata on the left and the grammar on the right. You are free to edit the grammar yourself or let JFLAP more or less do the work.

The *What's Left?* option will show which transition that not have been used in the grammar yet. JFLAP automatically puts labels under states to tell which symbols they represent in the grammar.



As mentioned you can either edit the right side grammar table or click on states to automatically reveal the grammar for each step. The *Hint* reveals which state you should select next. *Show All* automatically creates the grammar for you.



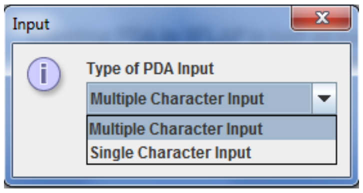
LHS		RHS
C	→	λ
S	→	0A
A	→	1S
B	→	1C
C	→	0C
D	→	1C
S	→	0D
A	→	0B
B	→	A

Once the grammar is complete you can select *Export* to open a new JFLAP window with your new Grammar, don't forgot to save if you want your grammar saved.

Context Free Grammar and Pushdown Automata

Pushdown Automata

Creating PDA in JFLAP is just as easy as creating FA but there are some differences. First you select "Pushdown Automaton" in the selection menu which also shows up when selecting "new" in File.

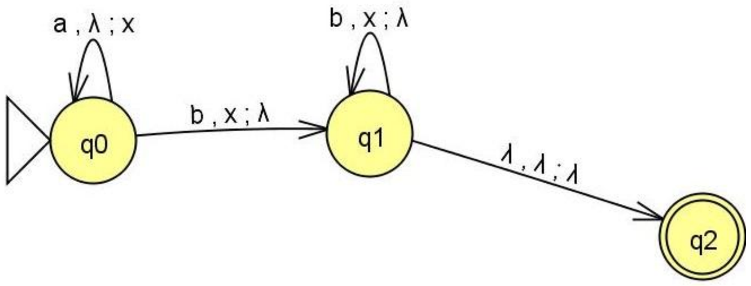

Input

Type of PDA Input

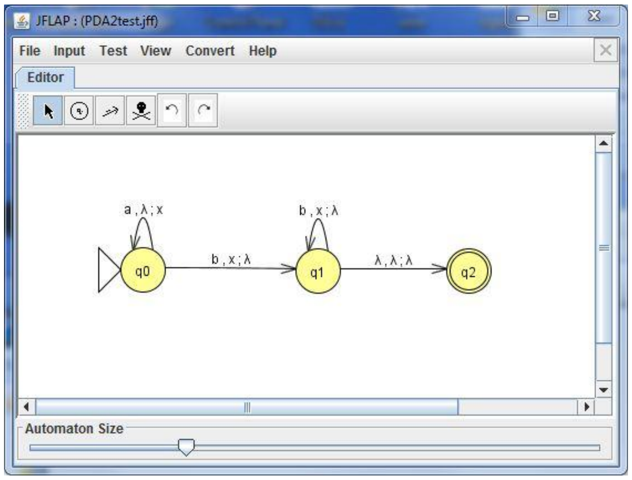
Multiple Character Input

Multiple Character Input
Single Character Input

First the program asks if you want to use either Multiple or Single Character Input which means how many characters may be consumed at each step.

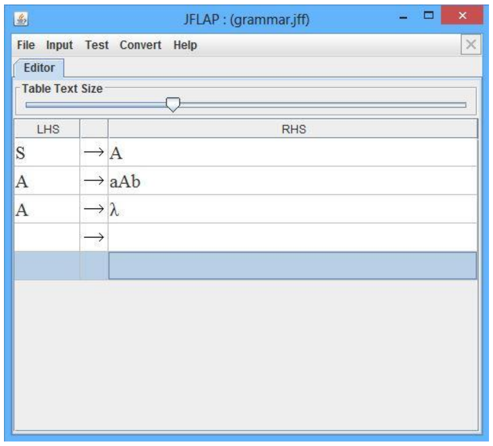


The editor looks exactly like editing Finite Automata. This PDA accepts the language L = {$a^n b^n$} where n > 0. The PDA is tested exactly like FA's. Transitions are different where they now consist of three variables. First is the input, second pops from the stack and the third variable is push.
Try to debug the PDA with the step by state feature. You will notice that you have a stack that is added and removed from during each step.



Grammar

Grammar is created using a table in JFLAP. If you choose grammar as a new project in JFLAP you will have a table where you can edit the grammar. Here we have a simple grammar with the language $a^n b^n$ where n ≥ 0.

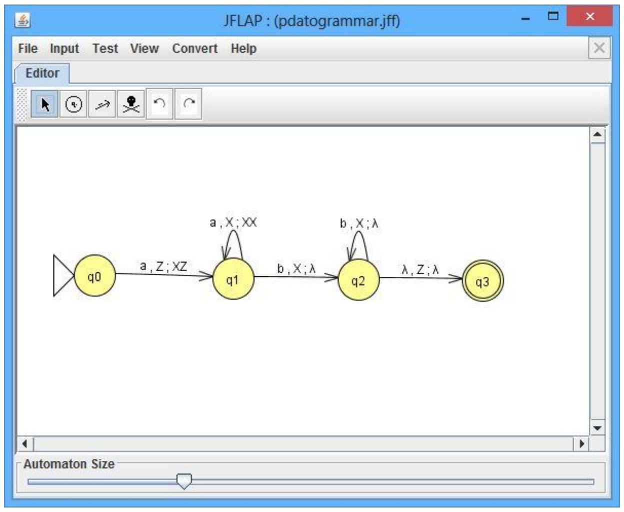


LHS		RHS
S	→	A
A	→	aAb
A	→	λ
	→	

Create and edit a grammar is simple. Create symbols by adding them to the LHS column and rules under RHS then press enter. The arrow between the symbols and rules is automatically created. Lambda is done by leaving the rule empty then press enter and you will see the lambda symbol there. Grammar can be tested by using any of the features under Input. One is Brute Force Parse which can test single strings each time.

Convert Pushdown Automata to Grammar

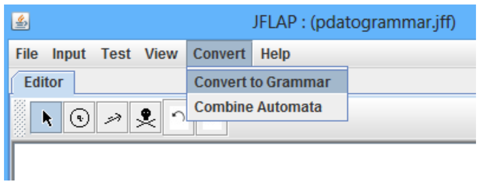
Conversion with PDA is very similar to converting FA, but there are differences that are good to know before creating PDA's.



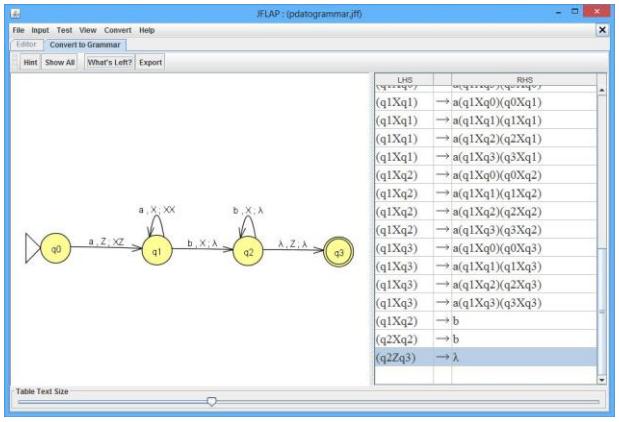
To convert PDA to Grammar in JFLAP, a couple of conditions must be met:

- For each transition, pop 1 symbol and push either 0 or 2 symbols.
- There must be only one final state with transitions that pop Z off the stack.

Trying to convert a PDA that not follows this will result in an error message that shows which transitions should be corrected. To start converting select Convert →Convert To Grammar.

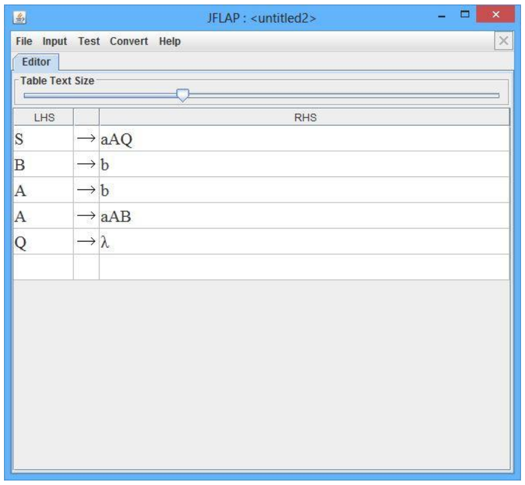


This is the conversion screen, on the left is the PDA and on the right is the grammar. You can either fill the grammar yourself or select each transition to fill out the grammar. Note that the grammar will be filled with a lot of useless rules which happens when using brute force.



LHS		RHS
(q1Xq1)	→	a(q1Xq0)(q0Xq1)
(q1Xq1)	→	a(q1Xq1)(q1Xq1)
(q1Xq1)	→	a(q1Xq2)(q2Xq1)
(q1Xq1)	→	a(q1Xq3)(q3Xq1)
(q1Xq2)	→	a(q1Xq0)(q0Xq2)
(q1Xq2)	→	a(q1Xq1)(q1Xq2)
(q1Xq2)	→	a(q1Xq2)(q2Xq2)
(q1Xq2)	→	a(q1Xq3)(q3Xq2)
(q1Xq3)	→	a(q1Xq0)(q0Xq3)
(q1Xq3)	→	a(q1Xq1)(q1Xq3)
(q1Xq3)	→	a(q1Xq2)(q2Xq3)
(q1Xq3)	→	a(q1Xq3)(q3Xq3)
(q1Xq2)	→	b
(q2Xq2)	→	b
(q2Zq3)	→	λ

When all rules are set from every transition, you can choose to select Export to transfer the Grammar to a new JFLAP window. JFLAP can trim down the useless rules into a readable grammar.

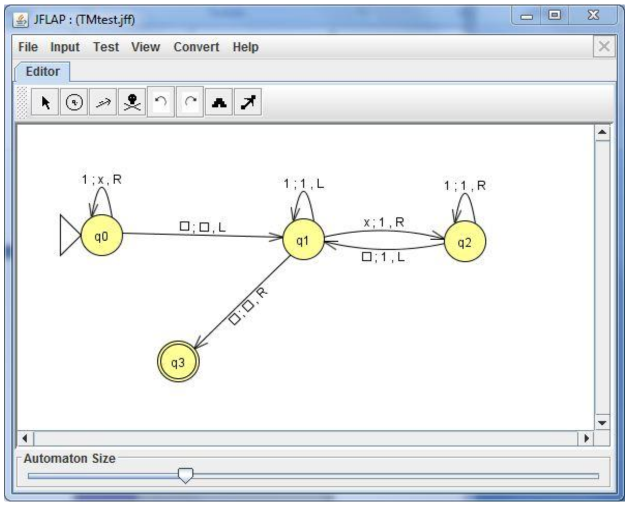


LHS		RHS
S	→	aAQ
B	→	b
A	→	b
A	→	aAB
Q	→	λ

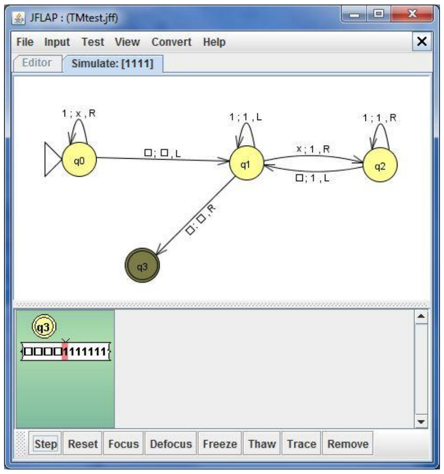
Restriction Free Languages and Turing Machines

Turing Machine

A Turing machine uses a tape that can be written and removed to by the head. For every transition there are three variables to set. First one is what value is expected to be under the head. Second is if the first is correct what should be written under the head. Third and last is the direction the head should take, the head can go left (L), right (R) or stay (S). In JFLAP □ is the blank symbol. Create this Turing machine below which duplicates the number of ones on the tape.



When testing your Turing machine you can see the tape. Stepping trough updates the currently active state alongside the tape's content. In this example the tape starts with the string "1111", where the TM steps trough and changes each 1 to an x. To know how many extra 1 to write, x will act as a symbol for each extra 1 to be written. The TM stops once the computation is done the tape should contain the result with the head in its starting position.



8.2 Appendix 2. Assignments

JFLAP EXERCISES

A few words of introduction about JFLAP.

JFLAP is an automata simulator that supports DFA/NFA, PDA, Turing machines and more.
JFLAP also supports Regular Expressions and Grammar which can be converted to an automaton and back. JFLAP has been used worldwide on many automata theory courses and is still under development.

Links to manuals and resources.

If you have problems use the provided manual *JFLAP User Manual.*
JFLAP's home page also contains a very thorough tutorial of everything the program can do.
JFLAP Home Webpage: www.jflap.org
Recommended reading is: *JFLAP - An Interactive Formal*
Languages and Automata Package by Susan H. Rodger and Thomas W. Finley
ISBN: 9780763738341
Here are as well files for this book: http://www.cs.duke.edu/csed/jflap/jflapbook/files/

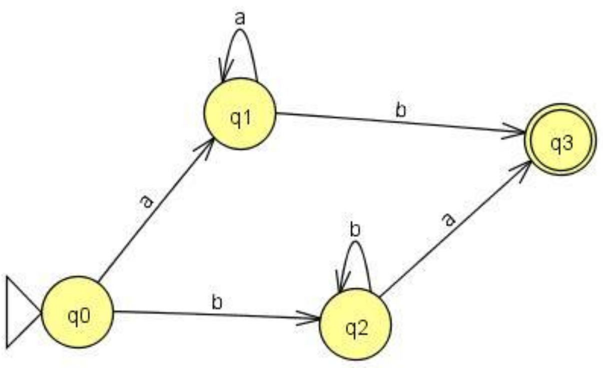
Lab Instructions

When you are done with each task, save your work in JFLAP as .jff files and call them ass1_1.jff after each assignment. Also a Word document should be provided if there are any questions asked. Save everything in a compressed RAR file with the name laborationX_yourname.rar.

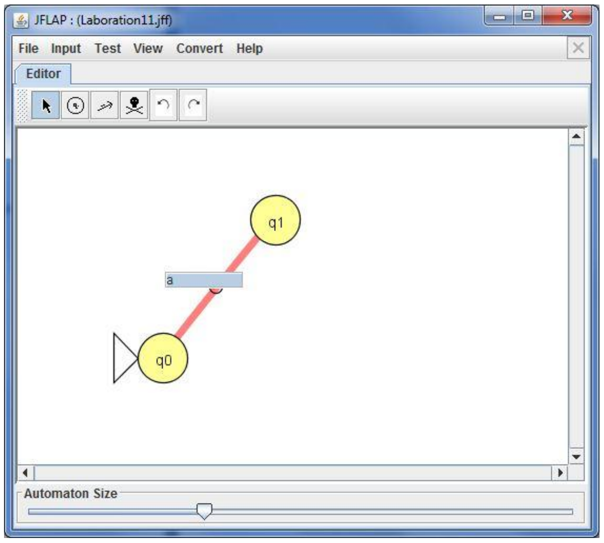
Assignment 1: JFLAP EXERCISES ON REGULAR LANGUAGES AND FINITE STATE AUTOMATA

Assignment 1.1

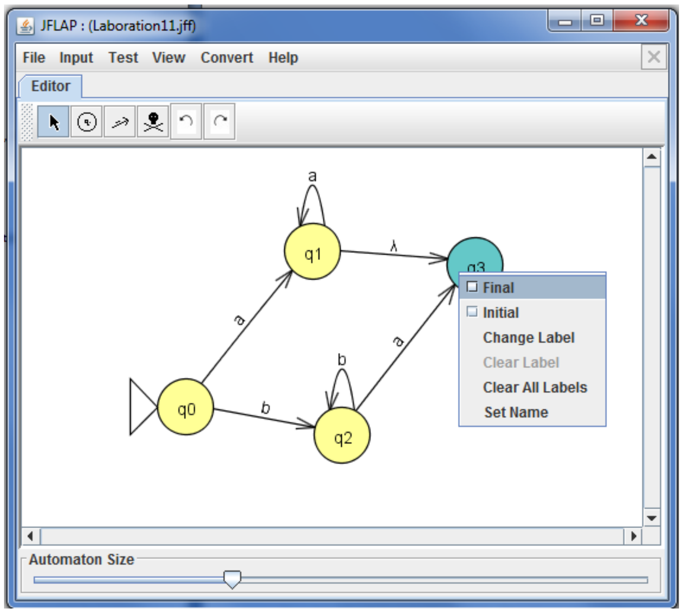
Create this Deterministic Finite Automaton. First start JFLAP and select “Finite Automaton” as a new automaton. Use the toolbar to drag and drop states and transitions.



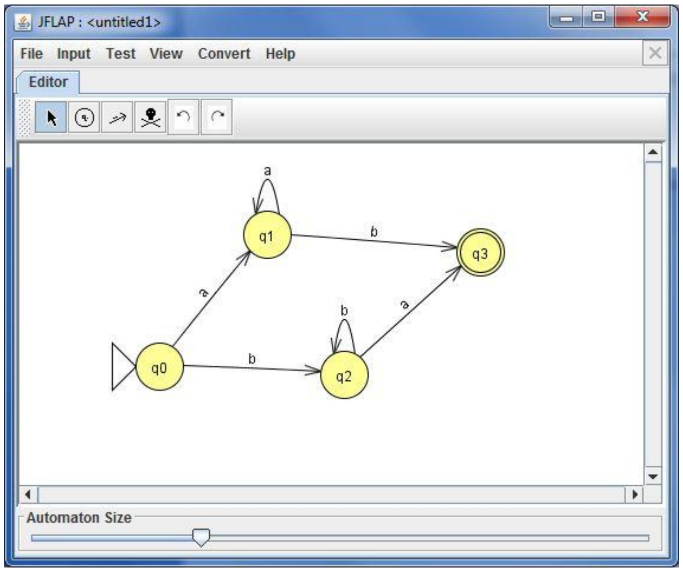
Select the transitions with the selection tool. Then type the input character for each transition.



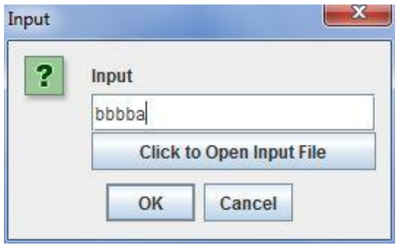
Remember that initial and final states must be set to make the automaton run. State q0 is set as the initial state which means the starting state. State q3 will be the final state which is a state that should be active once the string is accepted and consumed. There can be more than one final state.



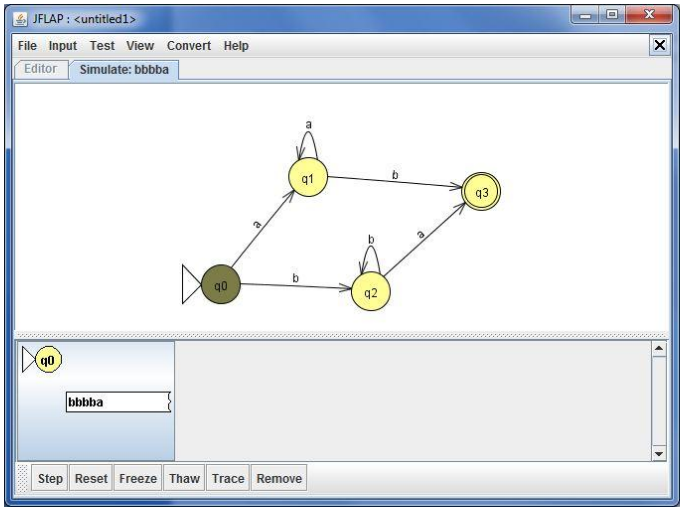
When you are done, the automaton should look like this.



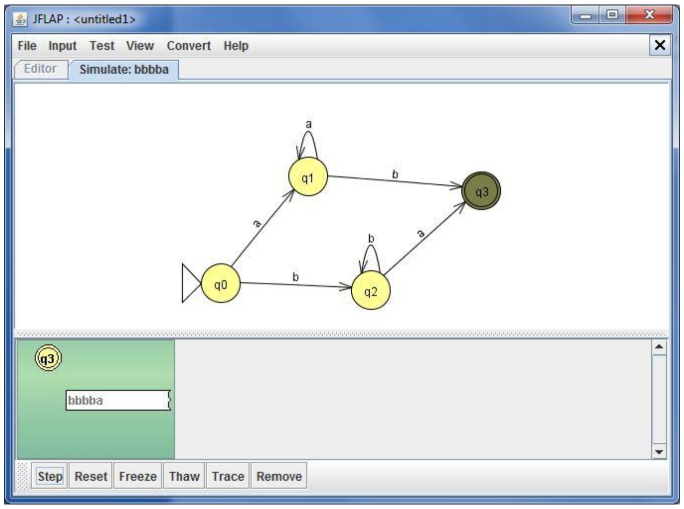
To run this DFA select *Input→Step By State*, then choose an input string to run the automata with.



You will see JFLAP's simulation view where you can debug the automaton while it consumes the input string. Use the Step button to step trough the string. Here the automaton have moved the active state and have consumed all the b characters.

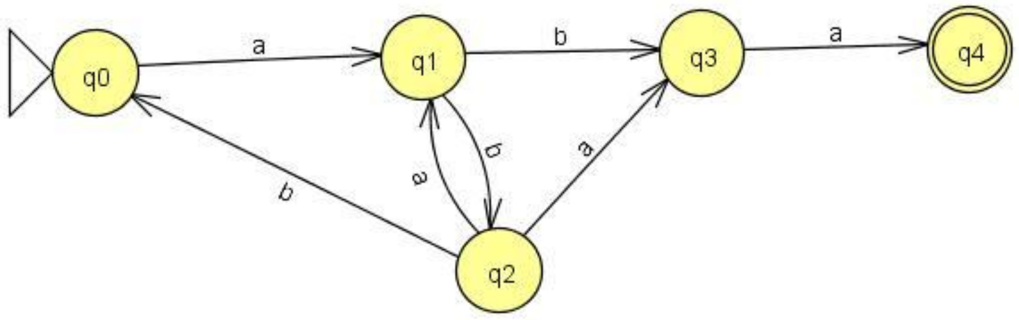

JFLAP : <untitled1>
File Input Test View Convert Help
Editor
Simulate: bbbba
a
q1
b
q3
a
b
a
q0
b
q2
q0
bbbba
Step Reset Freeze Thaw Trace Remove

Once the whole string is consumed the automata should park at the final state and the stack turns green. The string is accepted.


JFLAP : <untitled1>
File Input Test View Convert Help
Editor
Simulate: bbbba
a
q1
b
q3
a
b
a
q0
b
q2
q3
bbbba
Step Reset Freeze Thaw Trace Remove

Try to use Input $\rightarrow$ Multiple Run instead. Which strings are accepted and which ones are rejected?

Assignment 1.2
Create the following NFA:



a) Write down 4 accepted strings and 4 strings that are not accepted.

b) Which of the following regular languages over the alphabet Σ = {a,b} are accepted by the automaton?

   1) L = {aba},

   2) L = {aba*a},

   3) L = {a(ba)*ba},

   4) L = {a(bba)*baa}

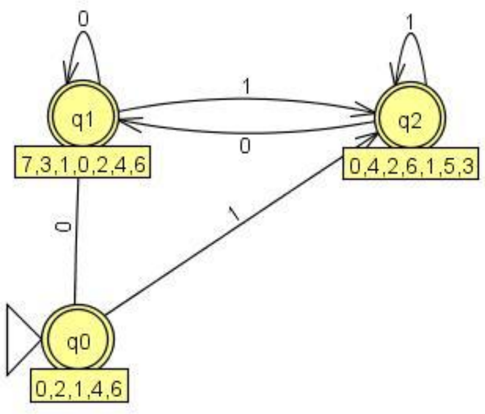
c) Construct a regular expression using JFLAP. Use Convert→Convert FA to RE.

d) Construct a Grammar using JFLAP, use Convert→Convert To Grammar.

e) Try to convert the automaton to a DFA. Once you are done JFLAP should export the result to a new JFLAP window. Save the resulting DFA as a new JFLAP file.

Assignment 1.3

JFLAP supports creation of FA from a regular expression. First you need to select “Regular Expression” in the selection menu. Just type the expression in the box, and then select ”Convert to FA” where you can step trough the conversion until the automaton can be exported to a new window. If the automaton is correct it should be possible to convert it back to a regular expression again. Use these regular expressions to be converted to a Finite Automaton.
Note that JFLAP creates a lot of lambda transitions in this part. The result can be improved by converting the resulting NFA to a DFA and then minimize the automaton.



Example, DFA created from the Regular Expression (1+0)*

Convert the following regular expressions to FA:

a) a(a+b)*b

b) 1((1+0)1)*

c) a(a+b)*bb(a+b)

d) ab(a+ba)*b*

Assignment 1.4

Create Regular Grammar and tell which language the grammar generates.
Then convert Regular Grammar to a Finite Automaton, try to do the FA yourself!
Then save the result.

a)
$$S \to aB$$
$$B \to bB$$
$$B \to \lambda$$

b)
$$S \to bA$$
$$A \to abB | baS$$
$$B \to bba$$

c)
$$S \to 10A$$
$$A \to 01B$$
$$A \to 11B$$
$$B \to 1S | 10A$$
$$B \to 11$$

Each of these Finite Automata can easily be converted back to a Grammar which you are free to try for yourself.

Assignment 1.5

a) Use this Regular Language to make a Nondeterministic Finite Automata.

Σ = {a,b,c} L = {aawbbwcc | w = {a,b,c}* | |w| is even }

After the NFA is made, find the Grammar for this NFA!

b) Now create a new FA(NFA or DFA) with this language:

Σ = {a,b} L = { ab(a(ba)*)* }

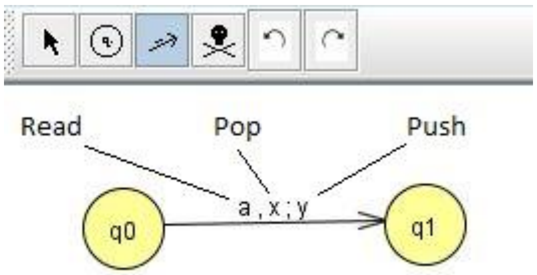
What is the minimal DFA for this language?

Save both the automaton and grammar. Languages should be saved as a Word document.

Assignment 2 JFLAP EXERCISES ON CONTEXT FREE LANGUAGES AND PUSHDOWN AUTOMATA

Assignment 2.1

Pushdown Automaton (PDA) is an automaton that also handles a stack. Unlike regular Finite Automata PDA's have a Pop and a Push symbol which handles the stack during execution.

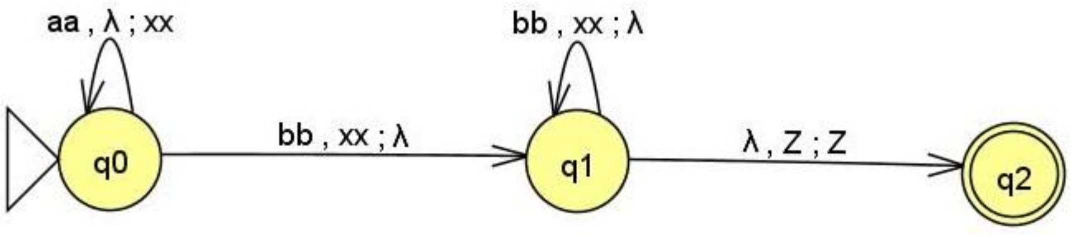


Create a PDA that accepts strings that contains the language L = {$a^x cb^{2x}$ | where x ≥ 0} using the alphabet ∑ = {a,b,c}.

Assignment 2.2

Create each PDA with at least five test results with the following languages over alphabet: Σ = {a,b}

a) Create the PDA from the Figure.



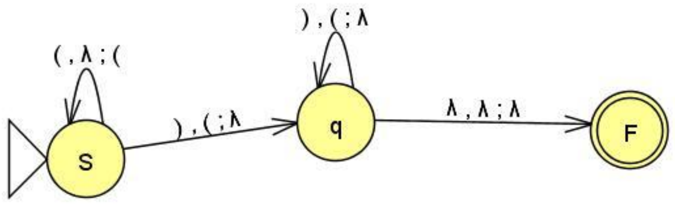
b) L = {$a^n b^n$ | where n > 0}

c) L = {$a^n b^n c^n$ | where n > 0}

d) L = {$W_1 c W_2$ | W_1, W_2 ε(a,b)* where $W_1 \neq W_2$}

Assignment 2.3

Consider the following Pushdown Automaton.



a) What language does this PDA accept?

b) Extend this automata to accept simple expressions like "((1+0) +1)". That is expressions with 1 and 0 and the + operator.

Assignment 2.4

Construct PDA that accepts the following grammars.

a)
$$S \to aBB$$
$$A \to cc$$
$$B \to bB|A$$
$$B \to bb$$
$$B \to \lambda$$

b)
$$S \to aABC$$
$$A \to aa|aaA$$
$$B \to bbB|bc$$
$$C \to ccC|c$$

c)
$$S \to AB$$
$$A \to aa|\lambda$$
$$B \to bbB|bb$$

Assignment 2.5

Consider the following language L = { $a^x b^{x2} a$ | where x > 0} using the alphabet Σ = { a,b }.

a) Create the Grammar for this language.

b) What type of grammar do you get? You can use JFLAP to determine which type you have via the Test menu.

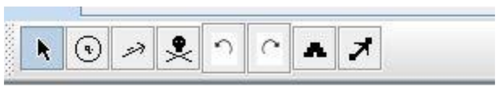
c) Create a PDA from this Grammar.

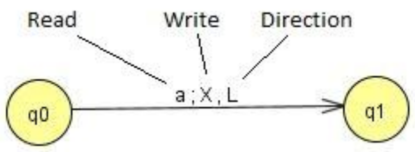
Save both the Grammar and the PDA in separate JFLAP files.

Assignment 3 JFLAP EXERCISES ON RESTRICTION FREE LANGUAGES AND TURING MACHINES

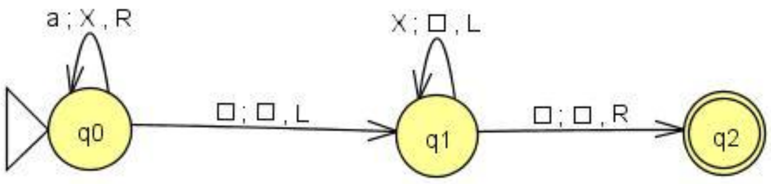
Assignment 3.1

Turing machines are more powerful than PDA and JFLAP gets very useful when creating these. Every transition has three symbols that you have to specify. Read, Write and Direction which tells the head what to do next. In JFLAP the blank symbol is represented as □.





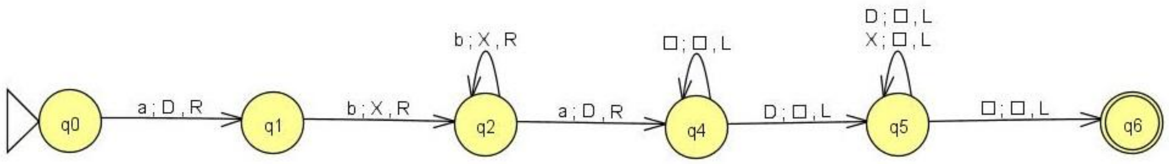
Now create the following TM in JFLAP.



a) What happens in this TM?
b) Why is it important to use a tape?
c) Show some input using JFLAPs transducer.
d) As you would notice empty tapes is accepted, how should you prevent that from happening? Save your project in a new file and create a TM that fixes that.

Assignment 3.2

Create the following Turing machine



Describe what this Turing machine does, recreate it and test run in JFLAP.
What language does this Turing machine accept?

Assignment 3.3

Construct a Turing machine that accepts the following languages.

a) L = {$a^n b^n$ | where n > 0}, Σ = {a,b}

b) L = {$a^n b^n c^n$ | where n > 0}, Σ = {a,b,c}

c) TM that doubles the number of a in a string of a's. (Ex. aaa should be aaaaaa.)

Assignment 3.4

Create a Turing machine that adds two binary strings into one single binary.
The operands should be binary strings with a plus operator in-between (Example is that 1010+1001 on the tape should produce the result 10011 on the tape).

To get you started the program can check each digit on both strings from left to right, and check what the result should be from these. Symbols can also be set (like 1010+1001B) at the end of the right operand where the program can write the answer.

Assignment 3.5

Create a Turing machine that accepts the language L = {$a^m b^n$ | m > n > 0}.