

# **LABORATORY MANUAL**

**Department of Computer Science and Engineering**

## LIST OF EXPERIMENTS

Expt. No.	Title of experiment	Corresponding CO
1.	About JFLAP	C 219.1
2.	Deterministic Finite Automata (DFA)	C 219.1
3.	Nondeterministic Finite Automata (NFA)	C 219.1
4.	Conversion of NFA to DFA	C 219.1
5.	DFA Minimization	C 219.1
6.	DFA to regular grammar conversion	C 219.3
7.	DFA to regular expression conversion	C 219.2
8.	Mealy machine	C 219.2
9.	Moore Machines	C 219.2
10.	Building Your First Pushdown Automaton	C 219.4
11.	Converting a NPDA to a Context-Free Grammar	C 219.4
12.	Single tape Turing machine	C 219.5
13.	Multi-tape Turing machine	C 219.5
14.	Converting Turing Machine to an Unrestricted Grammar	C 219.5
15.	Convert CFG to PDA (LL)	C 219.4
16.	Convert CFG to PDA (LR)	C 219.4
17.	Regular Expressions and Converting to a NFA	C 219.2
18.	Regular pumping lemma	C 219.2
19.	Context free pumping lemma	C 219.3
20.	Transform Grammar	C 219.3

<b>Content Beyond Syllabus</b>		
<b>21.</b>	Restriction Free Languages	C 219.5

## INTRODUCTION

In theoretical computer science and mathematics, the **theory of computation** is the branch of theoretical computer science that deals with how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory and languages, computability theory, and computational complexity theory, which are linked by the question: *"What are the fundamental capabilities and limitations of computers?"*.

Automata Theory is an exciting, theoretical branch of computer science. It established its roots during the 20th Century, as mathematicians began developing - both theoretically and literally - machines which imitated certain features of man, completing calculations more quickly and reliably. The word automaton itself, closely related to the word "automation", denotes automatic processes carrying out the production of specific processes. Simply stated, automata theory deals with the logic of computation with respect to simple machines, referred to as automata. Through automata, computer scientists are able to understand how machines compute functions and solve problems and more importantly, what it means for a function to be defined as computable or for a question to be described as decidable .

Automatons are abstract models of machines that perform computations on an input by moving through a series of states or configurations. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result, once the computation reaches an accepting configuration, it accepts that input. The most general and powerful automata is the Turing machine.

## PREFACE

The theory of computation is a branch of computer science and mathematics **combined** that "deals with how efficiently problems can be solved on a model of computation, using an algorithm". It studies the general properties of computation which in turn, helps us increase the efficiency at which computers solve problems.

## DO'S AND DONT'S

### DO's

1. Conform to the academic discipline of the department.
2. Enter your credentials in the laboratory attendance register.
3. Read and understand how to carry out an activity thoroughly before coming to the laboratory.
4. Ensure the uniqueness with respect to the methodology adopted for carrying out the experiments.
5. Shut down the machine once you are done using it.

### DONT'S

1. Eatables are not allowed in the laboratory.
2. Usage of mobile phones is strictly prohibited.
3. Do not open the system unit casing.
4. Do not remove anything from the computer laboratory without permission.
5. Do not touch, connect or disconnect any plug or cable without your faculty/laboratory Technician's permission.

## **GENERAL SAFETY INSTRUCTIONS**

1. Know the location of the fire extinguisher and the first aid box and how to use them in case of an emergency.
2. Report fire or accidents to your faculty /laboratory technician immediately.
3. Report any broken plugs or exposed electrical wires to your faculty/laboratory technician immediately.
4. Do not plug in external devices without scanning them for computer viruses.

## DETAILS OF THE EXPERIMENTS CONDUCTED

(TO BE USED BY THE STUDENTS IN THEIR RECORDS)

S. No	DATE OF CONDUCTION	EXPT. No	TITLE OF THE EXPERIMENT	PAGE No.	MARKS AWARDED (20)	FACULTY SIGNATURE WITH REMARK



## TAFL Lab (RCS 453)

<b>Name</b>	
<b>Roll No.</b>	
<b>Section- Batch</b>	

## INDEX

Experiment No.	Experiment Name	Date of Conduction	Date of Submission	Faculty Signature

## 1. JFLAP

JFLAP (Java Formal Languages and Automata Package) is interactive educational software written in Java for experimenting with topics in the computer science area of formal languages and automata theory, primarily intended for use at the undergraduate level or as an advanced topic for high school. JFLAP is software for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar. [Click here](#) for more information on what one can do with JFLAP.

### How To Run JFLAP

JFLAP 4.0b10 requires the presence of Java 1.4 or later. If you do not have Java of this version or later, see [Sun's Java page](#) for a download for your OS.

Assuming Java is properly installed on your system, running JFLAP requires only the presence of the JFLAP.jar file.

Here are instructions on how to run JFLAP once Java has been properly installed on several platforms:

**Mac OS X** Double click on the JFLAP.jar file.

**Unix/Linux** From the directory with the JFLAP.jar, execute ***java -jar JFLAP.jar*** from your favourite shell.

**Windows** Double click on the JFLAP.jar file.

**If that doesn't work**, then these steps:

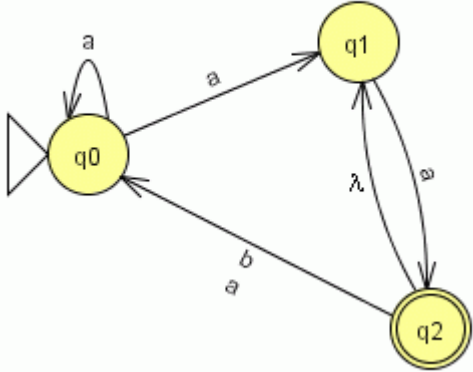
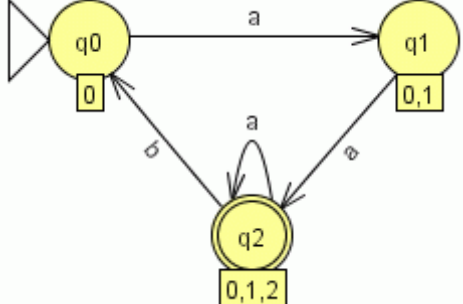
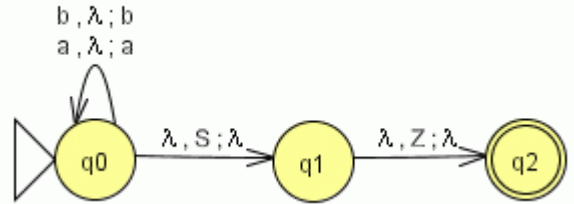
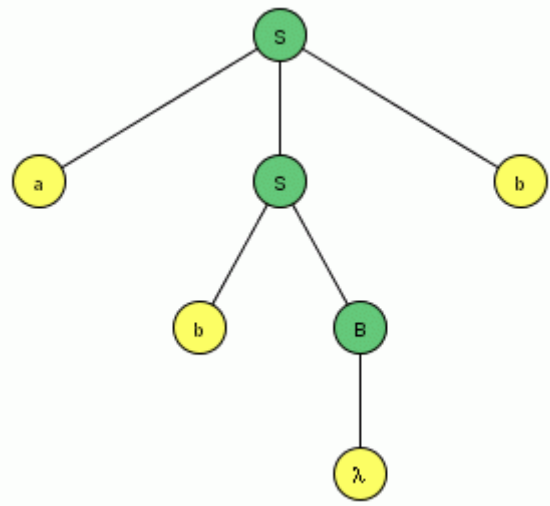
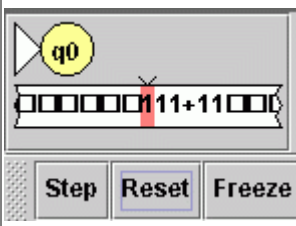
- 1) Make sure you have Java 1.4 installed
- 2) If when you download it, it tries to change its name to JFLAP.zip, save it, then rename it to JFLAP.jar
- 3) Now try double clicking on JFLAP.jar.

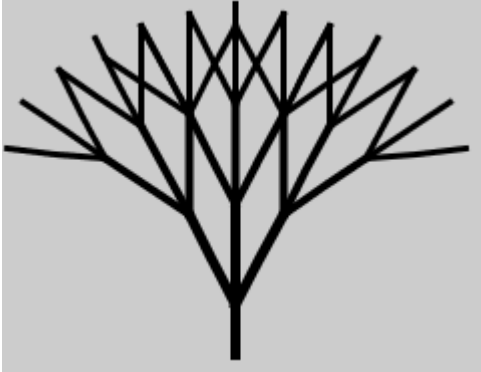
## **What Is JFLAP?**

**JFLAP (Java Formal Languages and Automata Package)** is a package of graphical tools which can be used as an aid in learning the basic concepts of Formal Languages and Automata Theory.

## **JFLAP Versions**

- Jan 22, 2003 - first released
- Jan 24, 2003
- Feb 3, 2003
- Mar 5, 2003
- May 22, 2003
- June 18, 2003
- July 22, 2003
- Jan 2, 2004
- Sep 14, 2004
- Jan 15, 2005
- Feb 12, 2005 - Now JFLAP works in Java 5.0! - VERY STABLE VERSION
- Feb 23, 2006 - JFLAP with Building Blocks - BETA VERSION
- Sept. 12, 2006 - JFLAP Version 6.0
- May 18, 2007 - JFLAP Version 6.1
- Jan 14, 2008 - JFLAP Version 6.2
- May 24, 2008 - JFLAP Version 6.3
- July 13, 2008 - JFLAP Version 6.4
- August 28, 2009 - JFLAP Version 7.0
- October 11, 2009 - JFLAP Version 7.0 (with one bug fix)
- May 15, 2011 - JFLAP Version 7.0 (with SVG fixed and thinner JFLAP)
- Jan 24, 2015 - JFLAP Version 8.0 BETA (Not Stable)
- Jul 27, 2018 - JFLAP Version 7.1

<p><b>Regular languages - create</b></p> <ul style="list-style-type: none"> <li>• DFA</li> <li>• NFA</li> <li>• regular grammar</li> <li>• regular expression</li> </ul>	
<p><b>Regular languages - conversions</b></p> <ul style="list-style-type: none"> <li>• NFA <math>\rightarrow</math> DFA <math>\rightarrow</math> Minimal DFA</li> <li>• NFA <math>\leftrightarrow</math> regular expression</li> <li>• NFA <math>\leftrightarrow</math> regular grammar</li> </ul>	
<p><b>Context-free languages - create</b></p> <ul style="list-style-type: none"> <li>• push-down automaton</li> <li>• context-free grammar</li> </ul>	
<p><b>Context-free languages - transform</b></p> <ul style="list-style-type: none"> <li>• PDA <math>\rightarrow</math> CFG</li> <li>• CFG <math>\rightarrow</math> PDA (LL parser)</li> <li>• CFG <math>\rightarrow</math> PDA (SLR parser)</li> <li>• CFG <math>\rightarrow</math> CNF</li> <li>• CFG <math>\rightarrow</math> LL parse table and parser</li> <li>• CFG <math>\rightarrow</math> SLR parse table and parser</li> <li>• CFG <math>\rightarrow</math> Brute force parser</li> </ul>	
<p><b>Recursively Enumerable languages</b></p> <ul style="list-style-type: none"> <li>• Turing machine (1-tape)</li> <li>• Turing machine (multi-tape)</li> <li>• Turing machine (building blocks)</li> </ul>	

<ul style="list-style-type: none"> <li>• <b>unrestricted grammar</b></li> <li>• <b>unrestricted grammar</b> -&gt; <b>brute force parser</b></li> </ul>	
<p><b>L-Systems</b></p> <ul style="list-style-type: none"> <li>• <b>Create L-systems</b></li> </ul>	

## 2. Deterministic Finite Automata (DFA)

### Definition

JFLAP defines a finite automaton (FA)  $M$  as the quintuple  $M = (Q, \Sigma, \delta, q_s, F)$  where

$Q$  is a finite set of states  $\{q_i \mid i \text{ is a nonnegative integer}\}$

$\Sigma$  is the finite input alphabet

$\delta$  is the transition function,  $\delta : D \rightarrow 2^Q$  where  $D$  is a finite subset of  $Q \times \Sigma^*$

$q_s$  (is member of  $Q$ ) is the initial state

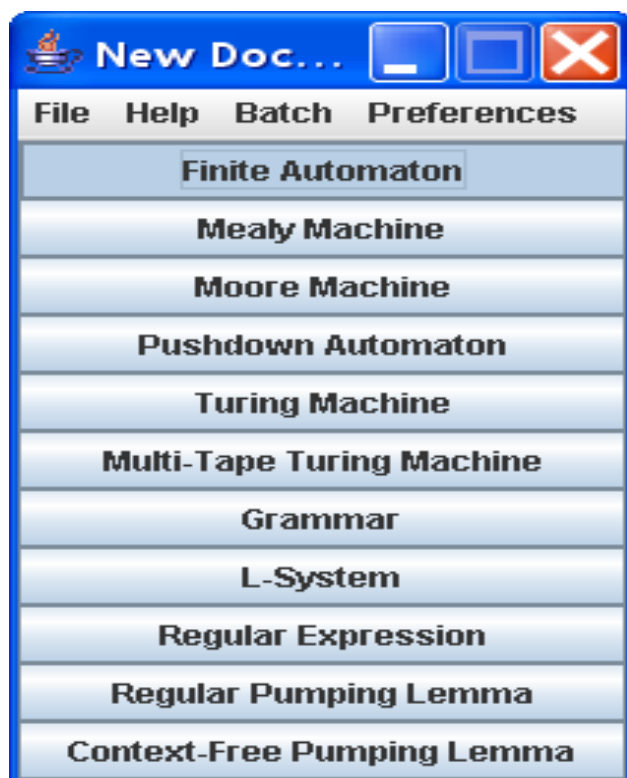
$F$  (is a subset of  $Q$ ) is the set of final states

Note that this definition includes both deterministic finite automata (DFAs), which we will be discussing shortly, and nondeterministic finite automata (NFAs), which we will touch on later.

Building the different types of automata in JFLAP is fairly similar, so let's start by building a DFA for the language  $L = \{a^m b^n : m \geq 0, n > 0, n \text{ is odd}\}$ . That is, we will build a DFA that recognizes that language of any number of  $a$ 's followed by any odd number of  $b$ 's. (Examples taken from *JFLAP: An Interactive Formal Languages and Automata Package* by Susan Rodger and Thomas Finley.)

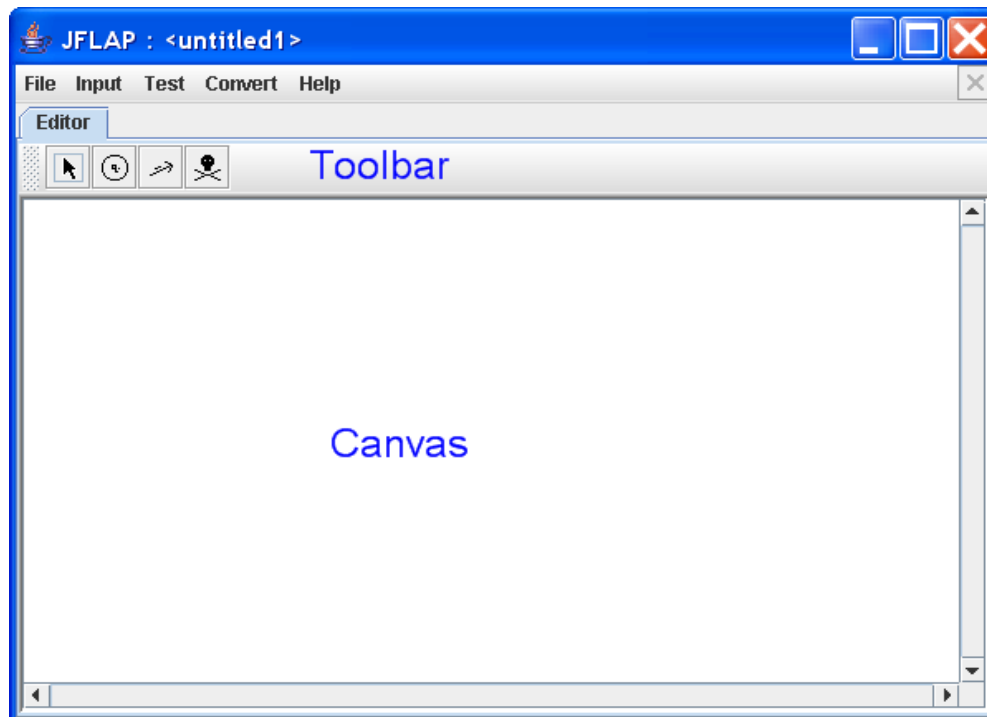
### The Editor Window

To start a new FA, start JFLAP and click the **Finite Automaton** option from the menu.



### Starting a new FA

This should bring up a new window that allows you to create and edit an FA. The editor is divided into two basic areas: the canvas, which you can construct your automaton on, and the toolbar, which holds the tools you need to construct your automaton.







The editor window


Let's take a closer look at the toolbar.



The FA toolbar

As you can see, the toolbar holds four tools:

- Attribute Editor tool  : sets initial and final states
- State Creator tool  : creates states
- Transition Creator tool  : creates transitions
- Deletor tool  : deletes states and transitions


To select a tool, click on the corresponding icon with your mouse. When a tool is selected, it is shaded, as the Attribute Editor tool  is above. Selecting the tool puts you in the corresponding mode. For instance, with the toolbar above, we are now in the Attribute Editor mode.

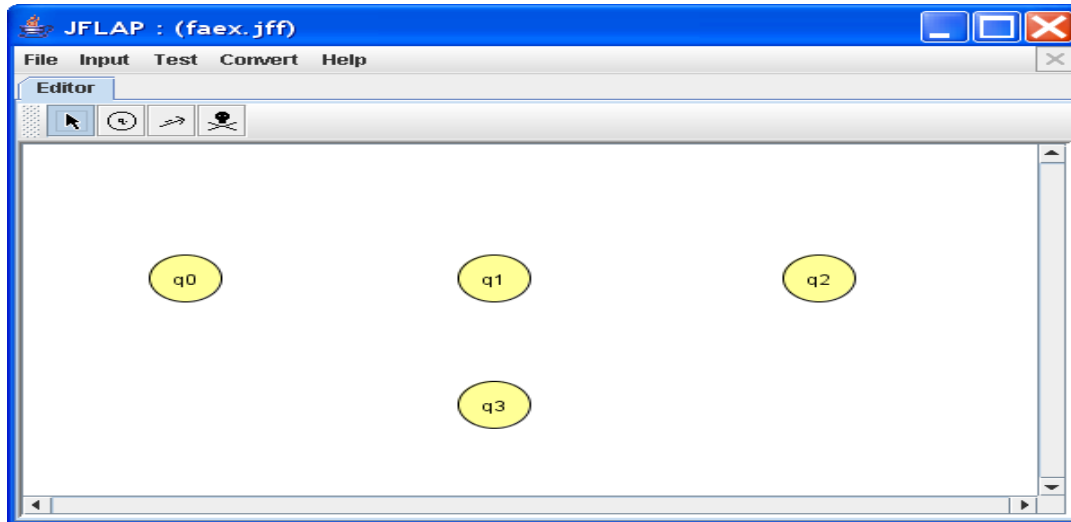
The different modes dictate the way mouse clicks affect the machine. For example, if we are in the State Creator mode, clicking on the canvas will create new states. These modes will be described in more detail shortly.

Now let's start creating our FA.



## Creating States


First, let's create several states. To do so we need to activate that State Creator tool by clicking the  button on the toolbar. Next, click on the canvas in different locations to create states. We are not very sure how many states we will need, so we created four states. Your editor window should look something like this:



States created

Now that we have created our states, let's define initial and final state.

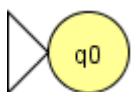
## Defining Initial and Final States

Arbitrarily, we decide that  $q_0$  will be our initial state. To define it to be our initial state, first select the Attribute Editor tool  on the toolbar. Now that we are in Attribute Editor mode, right-click on  $q_0$ . This should give us a pop-up menu that looks like this:



The state menu

From the pop-up menu, select the checkbox **Initial**. A white arrowhead appears to the left of  $q_0$  to indicate that it is the initial state.



$q_0$  defined as initial state

Next, let's create a final state. Arbitrarily, we select  $q_1$  as our final state. To define it as the final state, right-click on the state and click the checkbox **Final**. It will have a double outline, indicating that it is the final state.




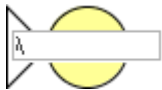
$q_1$  defined as final state

Now that we have defined initial and final states, let's move on to creating transitions.

### Creating Transitions

We know strings in our language can start with  $a$ 's, so, the initial state must have an outgoing transition on  $a$ . We also know that it can start with any number of  $a$ 's, which means that the FA should be in the same state after processing input of any number of  $a$ 's. Thus, the outgoing transition on  $a$  from  $q_0$  loops back to itself.

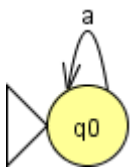
To create such a transition, first select the Transition Creator tool  from the toolbar. Next, click on  $q_0$  on the canvas. A text box should appear over the state:




Creating a transition

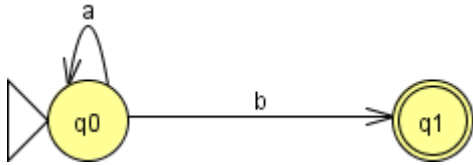
Note that  $\lambda$ , representing the empty string, is initially filled in for you. If you prefer  $\epsilon$  representing the empty string, select **Preferences : Preferences** in the main menu to change the symbol representing the empty string.

Type "a" in the text box and press **Enter**. If the text box isn't selected, press **Tab** to select it, then enter "a". When you are done, it should look like this:



Transition created

Next, we know that strings in our language must end with a odd number of  $b$ 's. Thus, we know that the outgoing transition on  $b$  from  $q_0$  must be to a final state, as a string ending with one  $b$  should be accepted. To create a transition from our initial state  $q_0$  to our final state  $q_1$ , first ensure that the Transition Creator tool  is selected on the toolbar. Next, click and hold on  $q_0$ , and drag the mouse to  $q_1$  and release the mouse button. Enter "b" in the textbox the same way you entered "a" for the previous transition. The transition between two states should look like this:

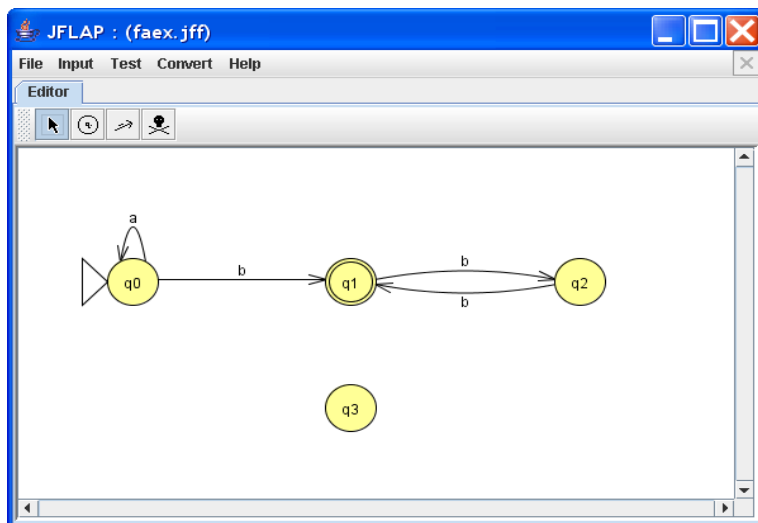


Second transition created

Lastly, we know that only strings that end with an odd number of  $b$ 's should be accepted. Thus, we know that  $q_1$  has an outgoing transition on  $b$ , that it cannot loop back to  $q_1$ . There are two options for the transition: it can either go to the initial state  $q_0$ , or to a brand new state, say,  $q_2$ .

If the transition on  $b$  was to the initial state  $q_0$ , strings would not have to be of the form  $a^m b^n$ ; strings such as  $ababab$  would also be accepted. Thus, the transition cannot be to  $q_0$ , and it must be to  $q_2$ .


Create a transition on  $b$  from  $q_1$  to  $q_2$ . As the FA should accept strings that end with an odd number of  $b$ 's, create another transition on  $b$  from  $q_2$  to  $q_1$ . Your FA is now a full, working FA! It should look something like this:

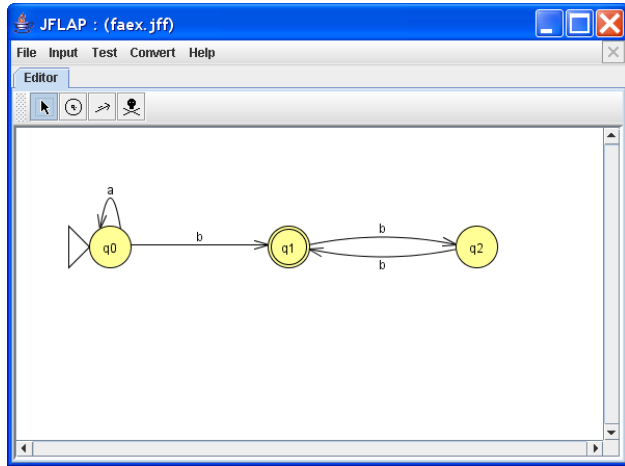


The working FA

You might notice that the  $q_3$  is not used and can be deleted. Next, we will describe how to delete states and transitions.

### Deleting States and Transitions

To delete  $q_3$ , first select the Deleter tool  on the toolbar. Next, click on the state  $q_3$ . Your editor window should now look something like this:



$q_3$  deleted

Similarly, to delete a transition, simply click on the input symbol of the transition when in Deletor mode.

Your FA, faex.jff, is now complete.

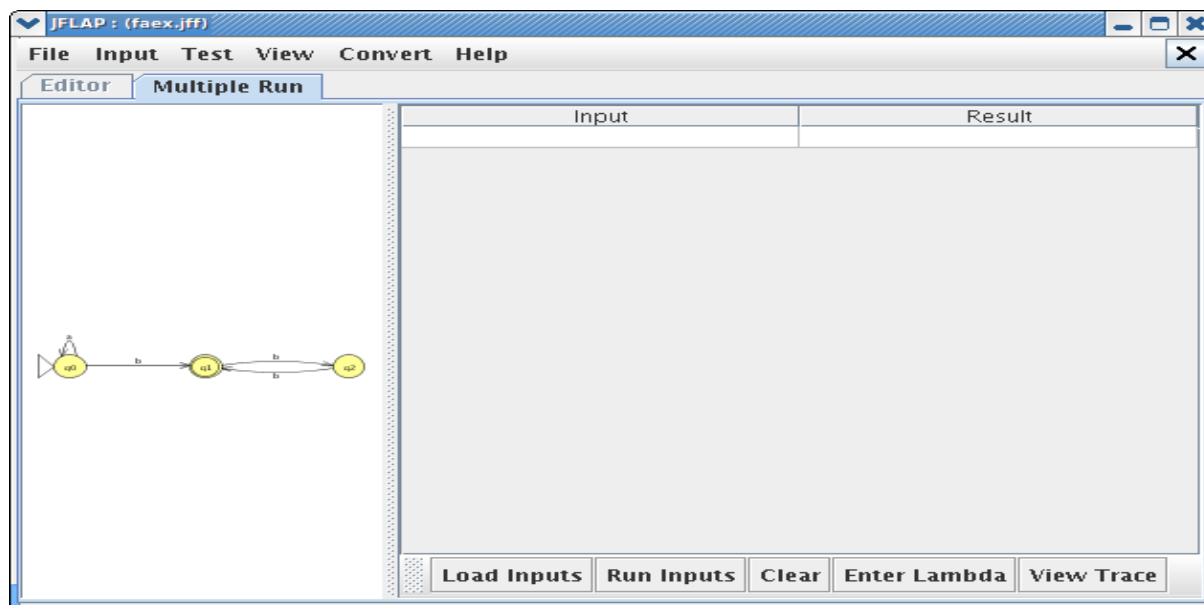
### Running the FA on Multiple Strings

Now that you've completed your FA, you might want to test it to see if it really accepts strings from the language. To do so, select **Input : Multiple Run** from the menu bar.



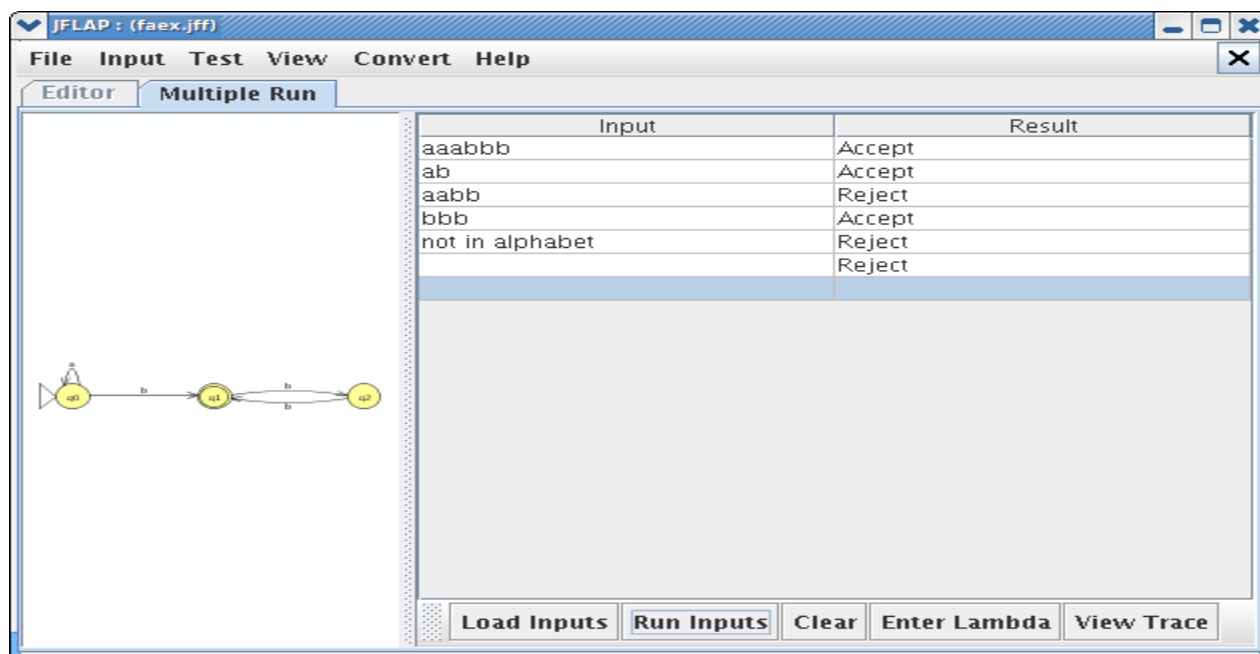
Starting a multiple run tab

A new tab will appear displaying the automaton on the left pane, and an input table on the right:



A new multiple run tab

To enter the input strings, click on the first row in the **Input** column and type in the string. Press **Enter** to continue to the next input string. When you are done, click **Run Inputs** to test your FA on all the input strings. The results, **Accept** or **Reject** are displayed in the **Result** column. You can also load the inputs from file delimited by white space. Simply click on **Load Inputs** and load the file to add additional input strings into multi-run pane.



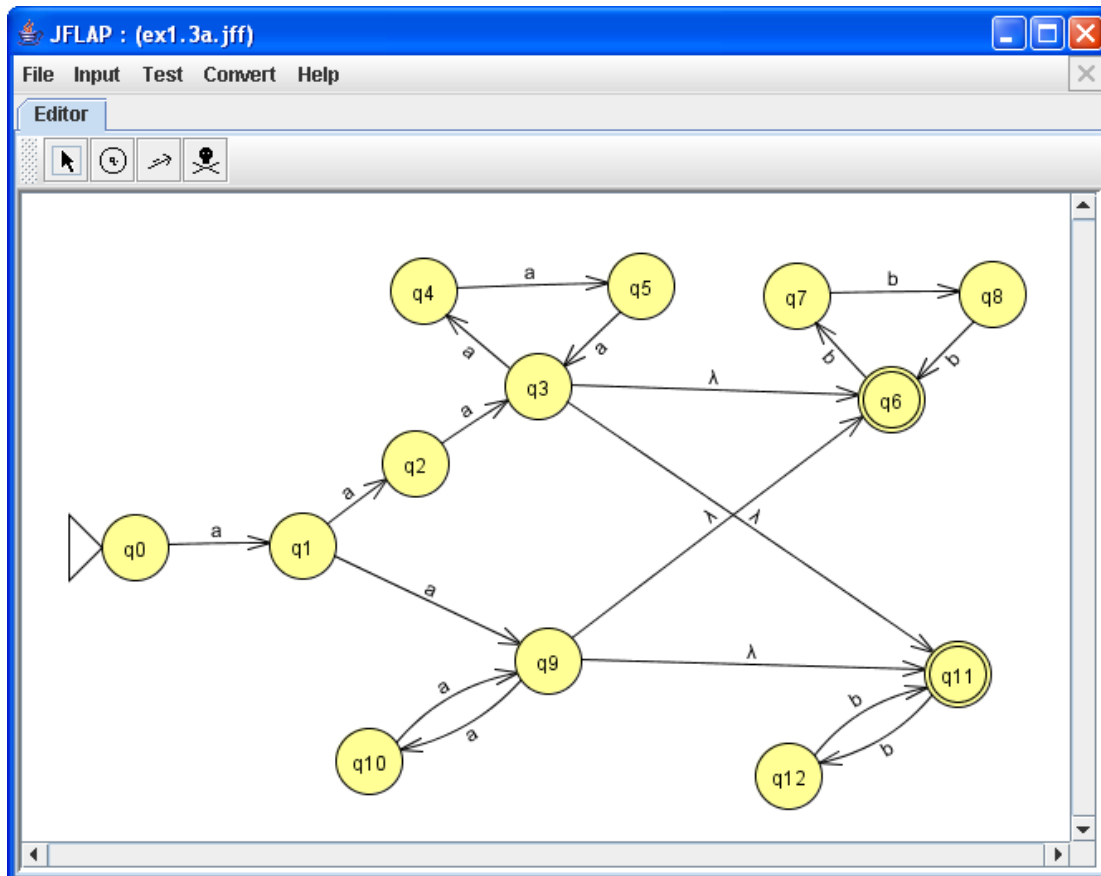
Running multiple inputs

Clicking **Clear** deletes all the input strings, while **Enter Lambda** enters the empty string at the cursor. **View Trace** brings up a separate window that shows the trace of the selected input. To return to the Editor window, select **File : Dismiss Tab** from the menu bar.

### 3. Nondeterministic Finite Automaton

Building a nondeterministic finite automaton (NFA) is very much like building a DFA. However, an NFA is different from a DFA in that it satisfies one of two conditions. Firstly, if the FA has two transitions from the same state that read the same symbol, the FA is considered an NFA. Secondly, if the FA has any transitions that read the empty string for input, it is also considered an NFA.

Let's take a look at this NFA, which can be accessed through [ex1.3a.jff](#): (Note: This example taken from *JFLAP: An Interactive Formal Languages and Automata Package* by Susan Rodger and Thomas Finley.)

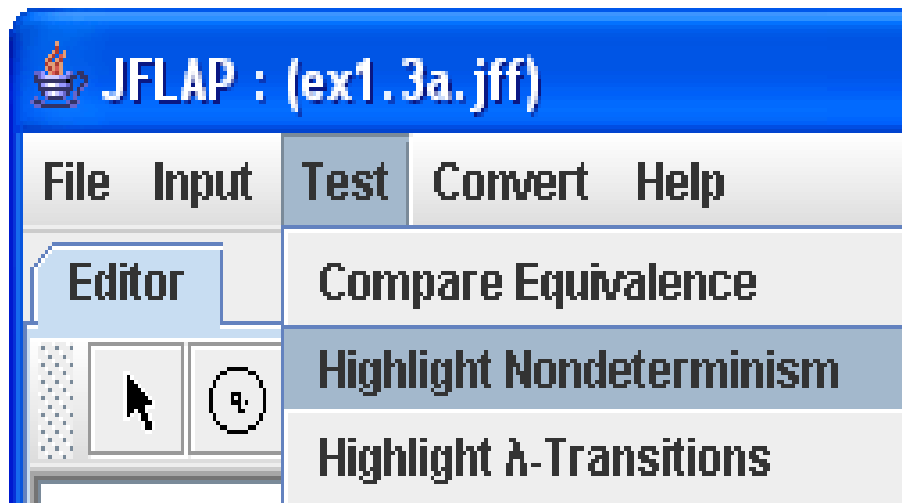


An NFA

We can immediately tell that this is an NFA because of the four  $\lambda$ -transitions coming from  $q_3$  and  $q_9$ , but we might not be sure if we have spotted all the nondeterministic states. JFLAP can help with that.

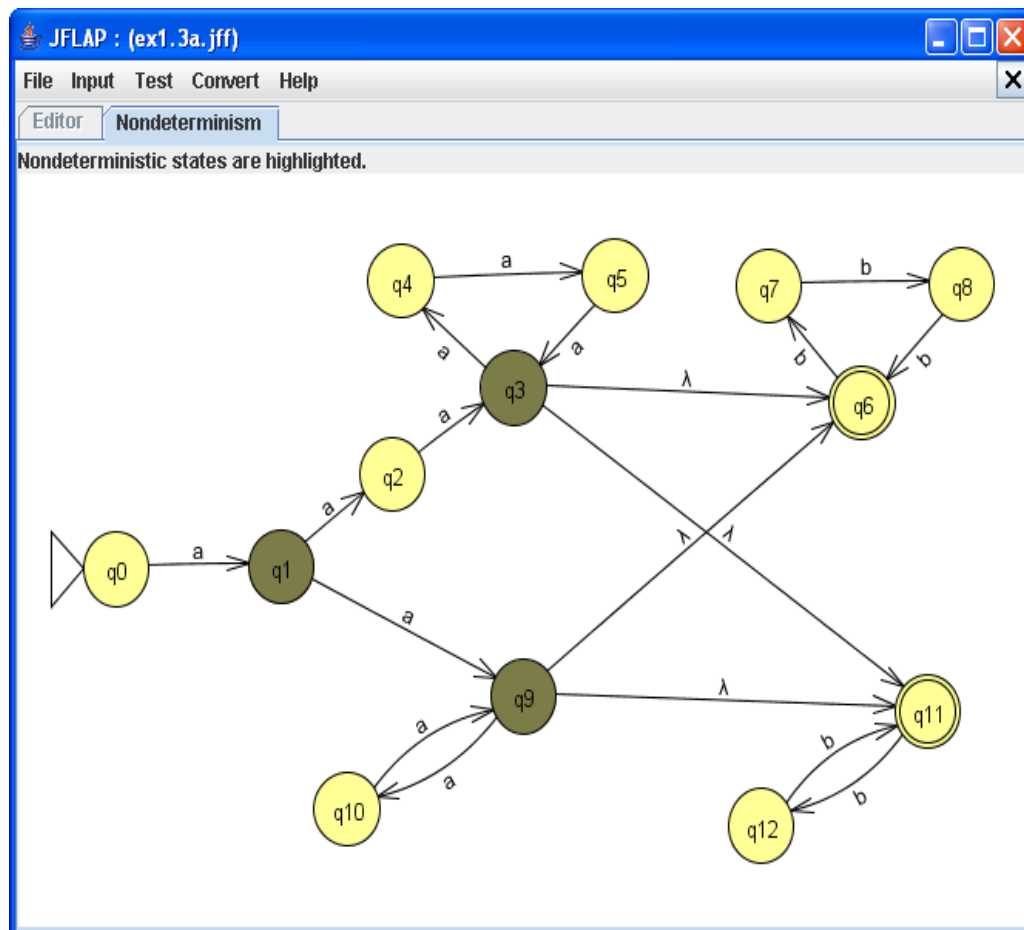
#### Highlighting Nondeterministic States

To see all the nondeterministic states in the NFA, select **Test : Highlight Nondeterminism** from the menu bar:



Highlighting nondeterministic states

A new tab will appear with the nondeterministic states shaded a darker color:



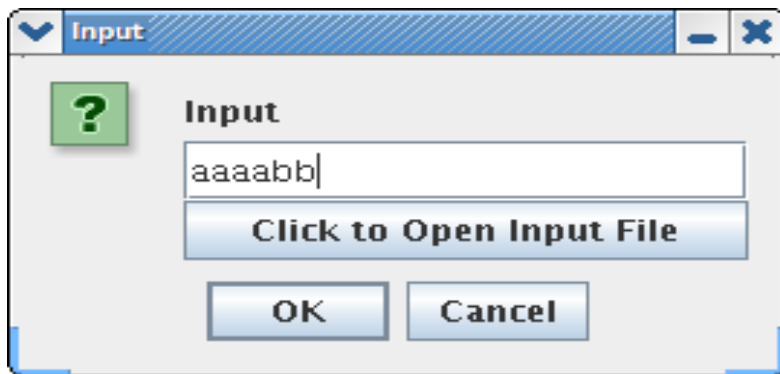
Nondeterministic states highlighted

As we can see,  $q_3$  and  $q_9$  are indeed nondeterministic because of their outgoing  $\lambda$ -transitions. Note that they would both be nondeterministic even if they each had one  $\lambda$ -transition instead of two: only one  $\lambda$ -transition is needed to make a state nondeterministic. We also see that  $q_1$  is nondeterministic because two of its outgoing transitions are on the same symbol,  $a$ . Next, we will go through JFLAP's tools for running input on an NFA.

## Running Input on an NFA

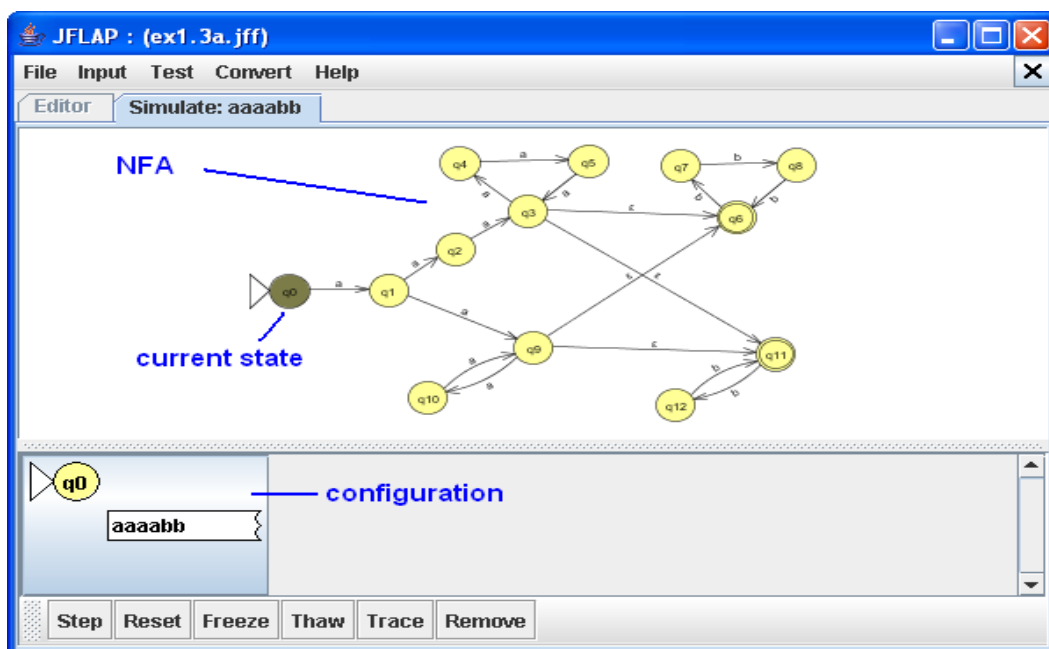
To step through input on an NFA, select **Input : Step with Closure...** from the menu bar. A dialog box prompting you for input will appear. Ordinarily, you would enter the input you wish to step through here. For now, type "aaaabb" in the dialog box and press **Enter**. You can also load the input file instead of typing the string.

**NOTE :** When loading input from the file, JFLAP determines end of input string by the white space. Thus if there is string "ab cd" in a file, only "ab" will be considered as an input ("cd" will be ignored since there is a white space before them).



Entering input

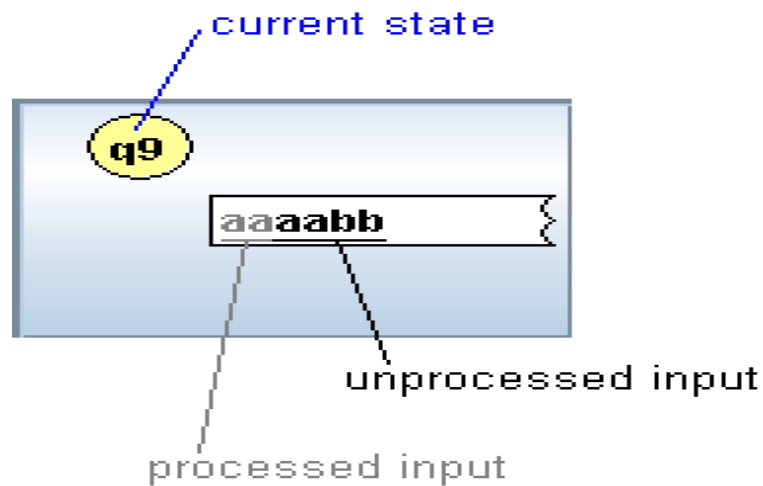
A new tab will appear displaying the automaton at the top of the window, and configurations at the bottom. The current state is shaded.



A new input tab

First, let's take a closer look at a configuration:

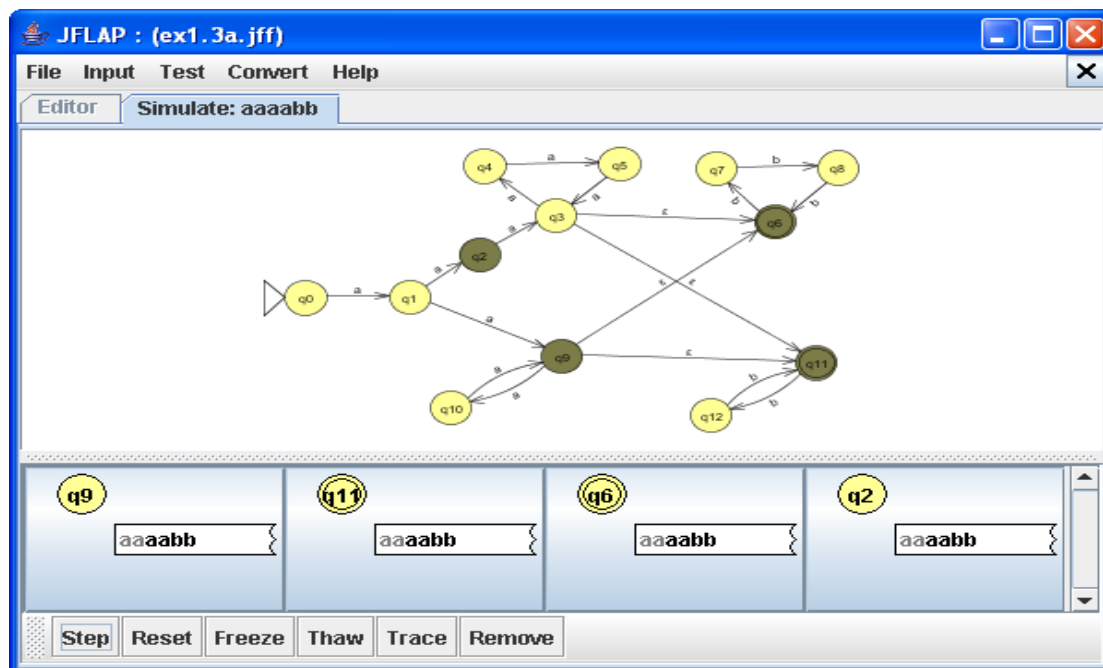




A configuration icon

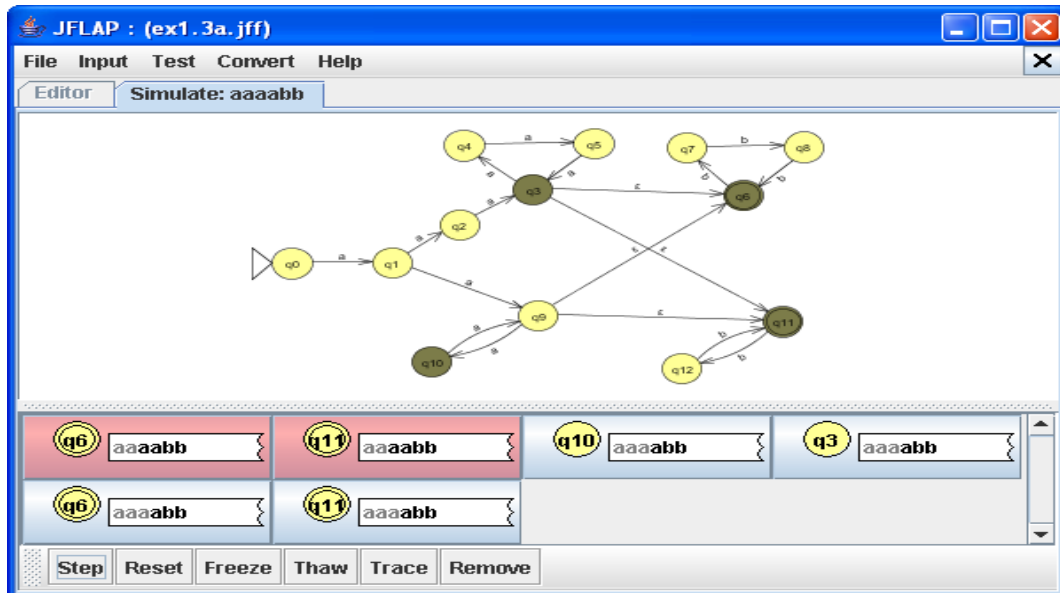
The configuration icon shows the current state of the configuration in the top left hand corner, and input on the white tape below. The processed input is displayed in gray, and the unprocessed input is black.

Click **Step** to process the next symbol of input. You will notice  $q_1$  becomes the shaded state in the NFA, and that the configuration icon changes, reflecting the fact that the first  $a$  has been processed. Click **Step** again to process the next  $a$ . You will find that four states are shaded instead of one, and there are four configurations instead of one.



$aa$  processed

This is because the machine is nondeterministic. From  $q_1$ , the NFA took both  $a$  transitions to  $q_2$  and  $q_9$ . As  $q_9$  has two  $\lambda$ -transitions (which do not need input), the NFA further produced two more configurations by taking those transitions. Thus, the simulator now has four configurations. Click **Step** again to process the next input symbol.

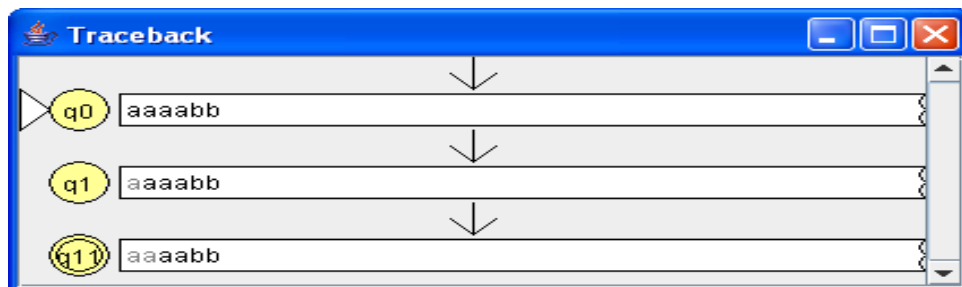


*aaa* processed

Notice that two of the configurations are highlighted red, indicating they were rejected. Looking at their input, we also know that only *aa* was processed. What happened?

### Producing a Trace

To select a configuration, click on it. It will become a solid color when selected, instead of the slightly graded color. Click on the icon for the rejected configuration with state  $q_{11}$ , and click **Trace**. A new window will appear showing the traceback of that configuration:



A configuration's traceback

The traceback shows the configuration after processing each input symbol. From the traceback, we can tell that that configuration started at  $q_0$  and took the transition to  $q_1$  after processing the first *a*. After processing the second *a*, it was in  $q_{11}$ . Although  $q_{11}$  is not adjacent to  $q_1$ , it can be reached by taking a  $\lambda$ -transition from  $q_9$ . As the simulator tried to process the next *a* on this configuration, it realized that there are no outgoing *a* transitions from  $q_{11}$  and thus rejected the configuration.

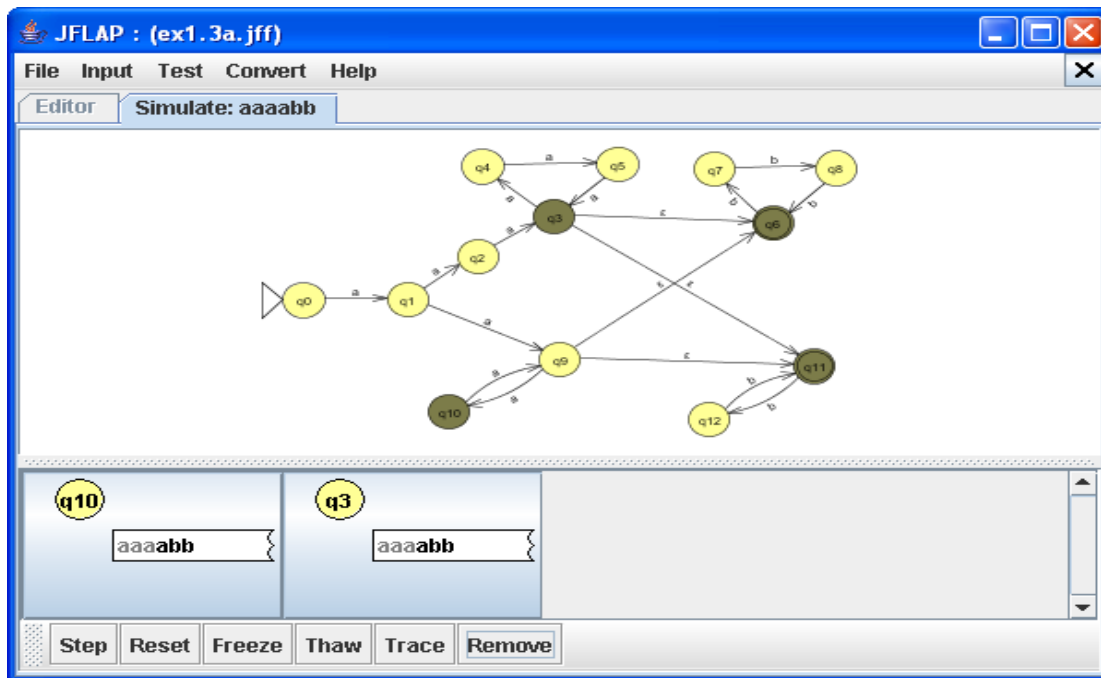
Although rejected configurations will remove themselves in the next step, we can also remove configurations that have not been rejected.

## Removing Configurations

Looking at the tracebacks of the rejected configurations, we can tell that any configurations that are in  $q_{11}$  or  $q_6$  and whose next input symbol is  $a$  will be rejected.

As the next input symbol is  $a$ , we can tell that the configurations that are currently in  $q_6$  and  $q_{11}$  will be rejected. Click once on each of the four configurations to select them, then click **Remove**. The simulator will no longer step these configurations. (Although we are only removing configurations that are about to be rejected, we can remove any configurations for any purpose, and the simulator will stop stepping through input on those configurations.)

Your simulator should now look something like this:



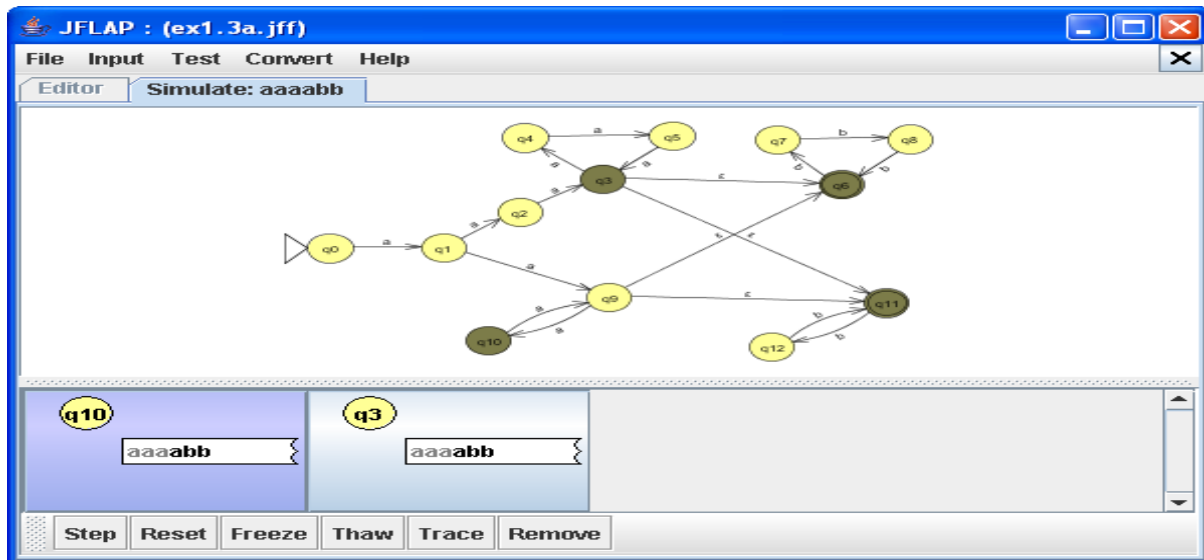
Rejected configurations removed

Now when we step the simulator, the two configurations will be stepped through.

Looking at the two configurations above, we might realize that the configuration on  $q_3$  will not lead to an accepting configuration. We can test our idea out by freezing the other configuration.

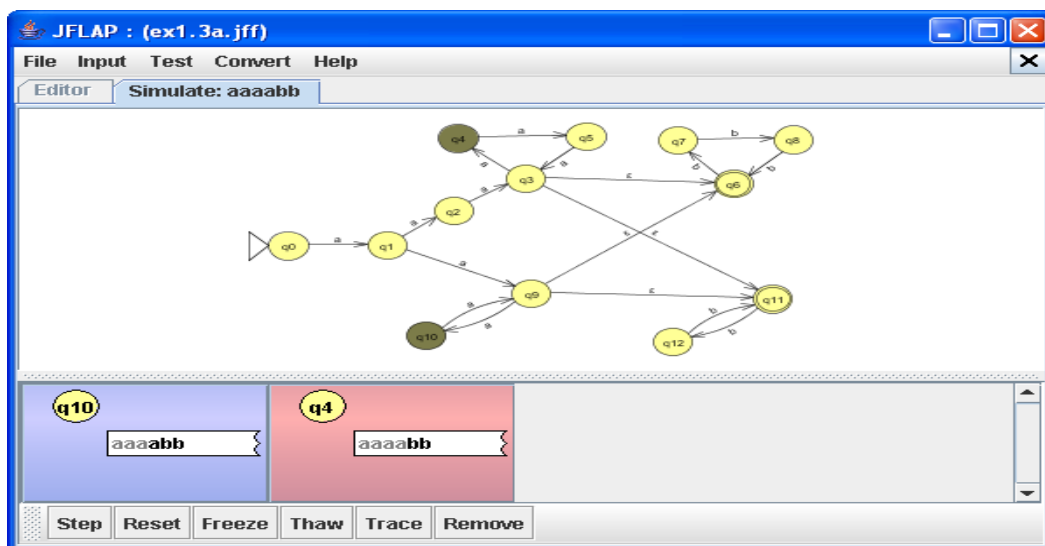
## Freezing and Thawing configurations

To freeze the configuration on  $q_{10}$ , click on  $q_{10}$  once, then click the **Freeze** button. When a configuration is frozen, it will be tinted a darker shade of purple:



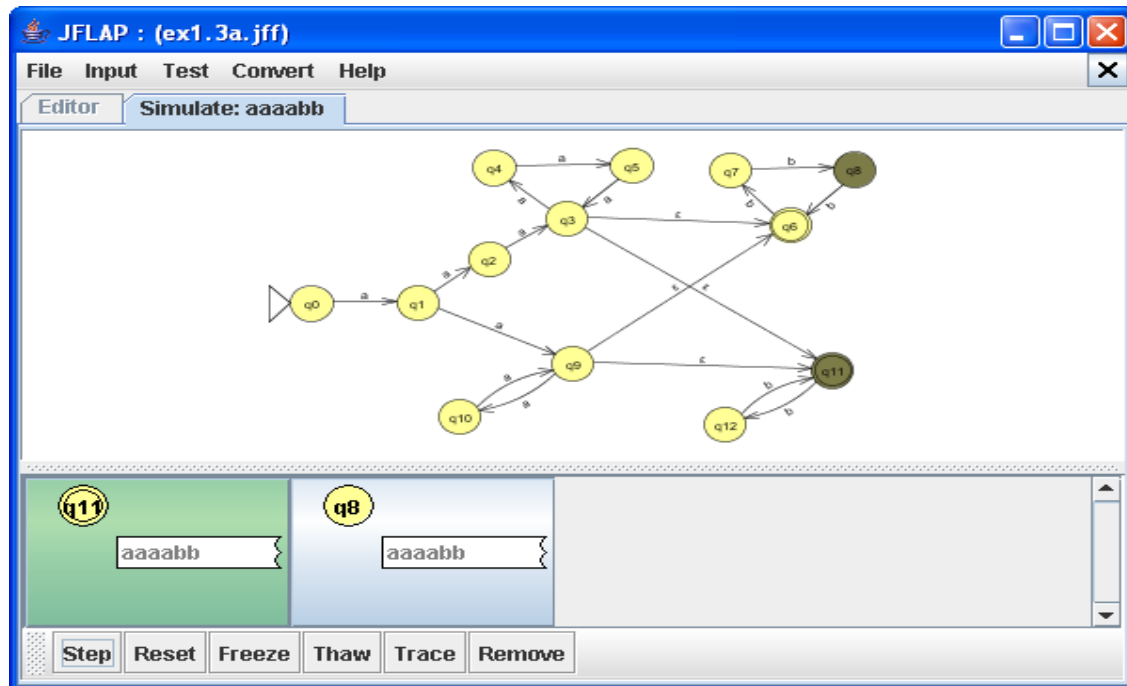
Configuration on  $q_{10}$  frozen

With that configuration frozen, as you click **Step** to step through the configuration on  $q_3$ , the frozen configuration remains the same. Clicking **Step** two more times will reveal that the configuration on  $q_3$  is not accepted either. Your simulator will now look like this:



Other configurations rejected

To proceed with the frozen configuration, select it and click **Thaw**. The simulator will now step through input as usual. Click **Step** another three times to find an accepting configuration. An accepting configuration is colored green:



Accepting configuration found

If we click **Step** again, we will see that the last configuration is rejected. Thus, there is only one accepting configuration. However, we might be unsure that this is really the case, as we had removed some configurations. We can double-check by resetting the simulator.

### Resetting the simulator

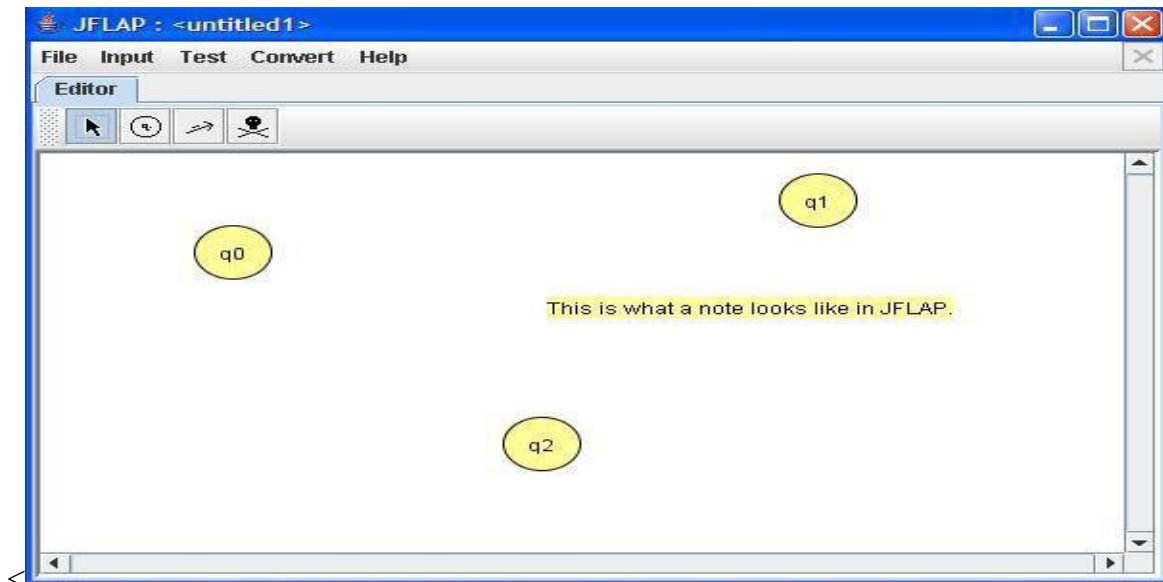
At any point in the simulation, we can restart the entire simulation process by clicking **Reset**. This will clear all the current configurations and restart the simulation. If we click **Reset** and step all the configurations, we will find that there is, indeed, only one accepting configuration.

This concludes the walkthrough, although there is an appendix noting a few more features that JFLAP supports.

## Appendix

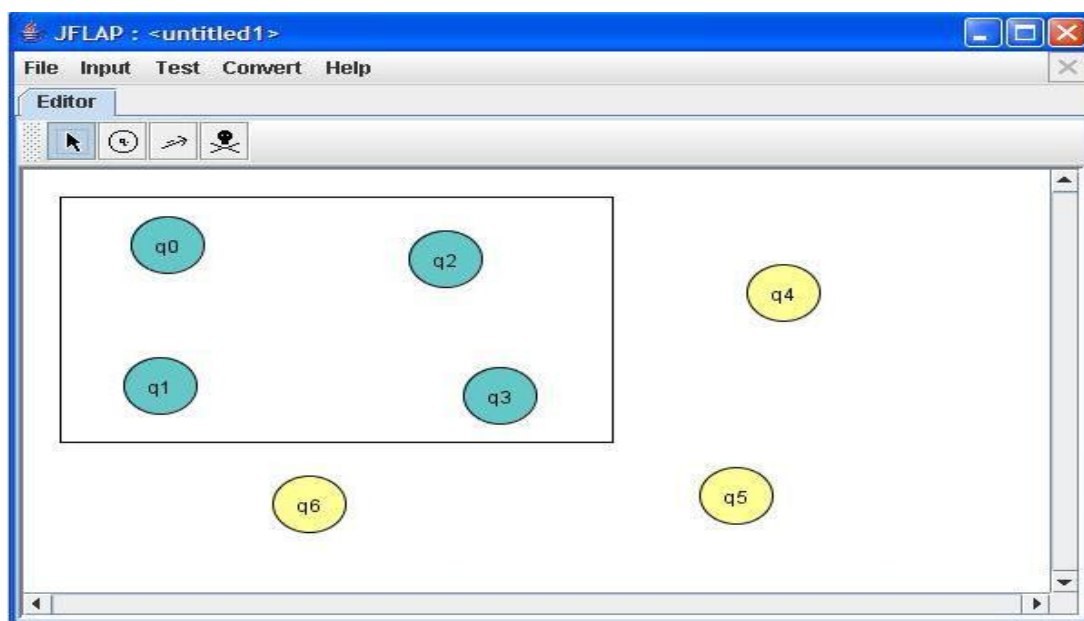
### Notes

To add a note to a JFLAP file, choose the Attribute Editor, right click and select "Add Note". The note will start with the message "insert\_text". To change the text simply click in the note, select where you want to start typing, and type your note. Click outside the note to get rid of the cursor. Click and drag the note to move it.



## Selection

To select more than one state or block at once, choose the attribute editor, click on empty space, and drag the mouse. A bounding box appears and all states and blocks within the box are selected, their color now blue. To move the selected states as a group, click and drag any of them. To deselect them, click anywhere else.

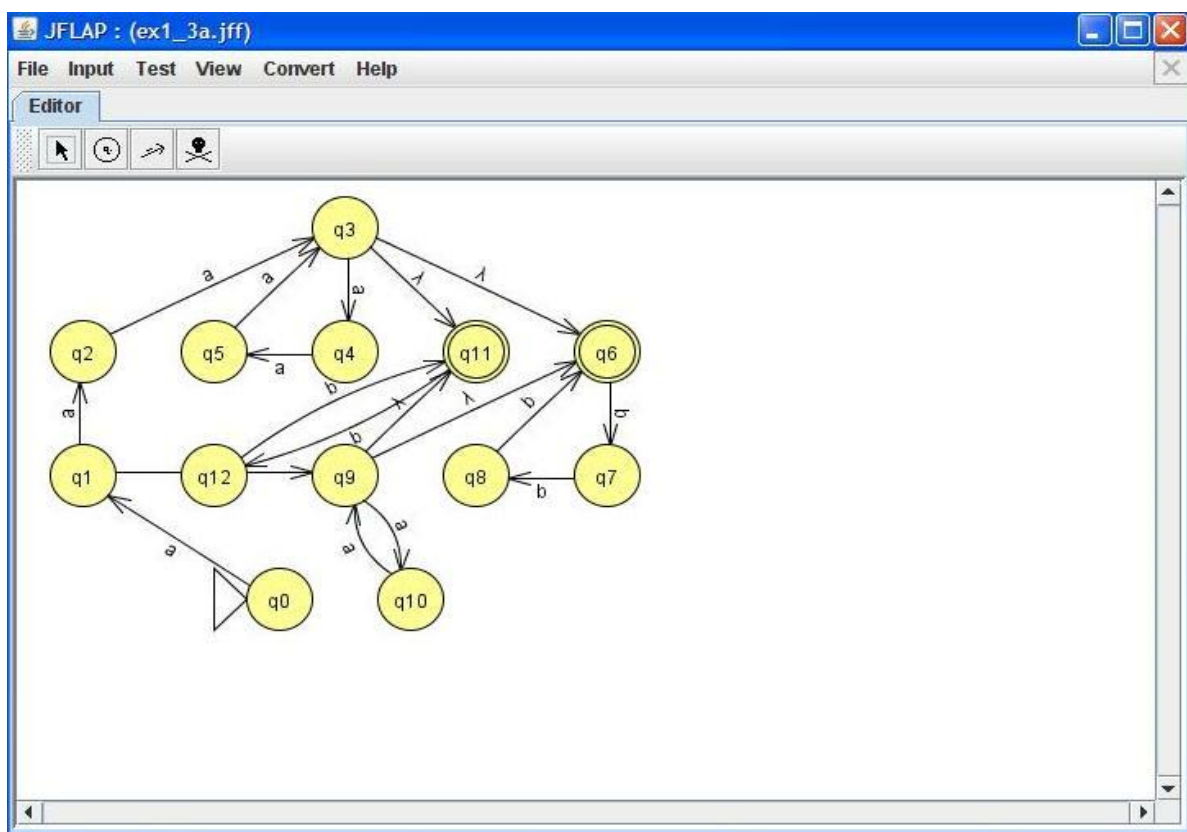
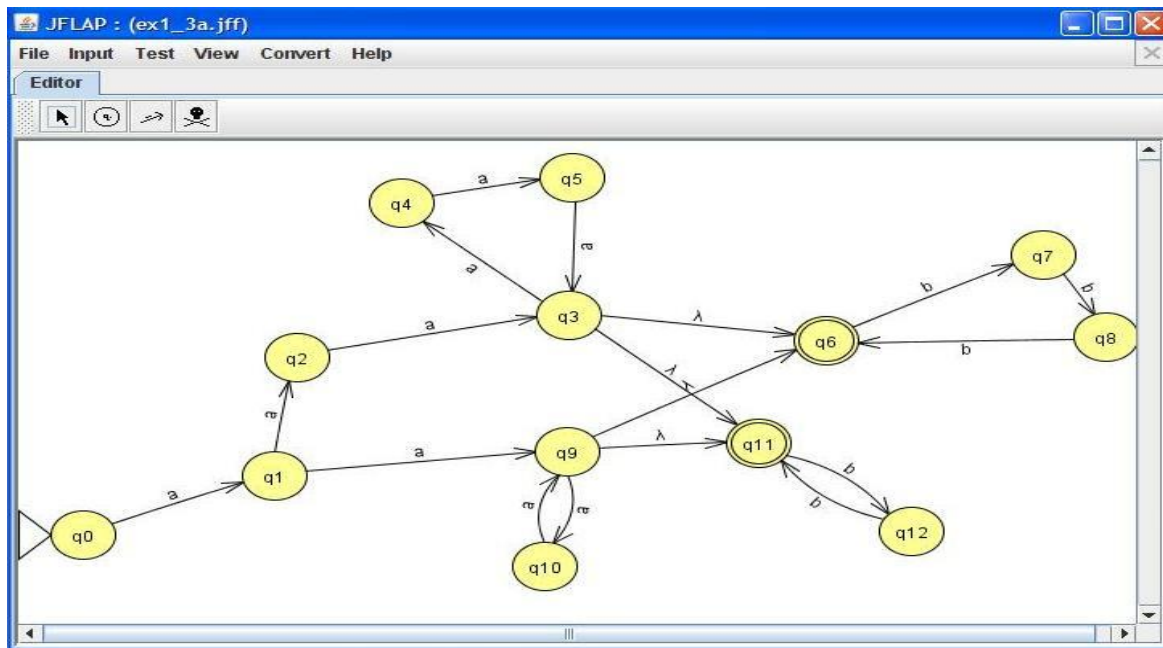


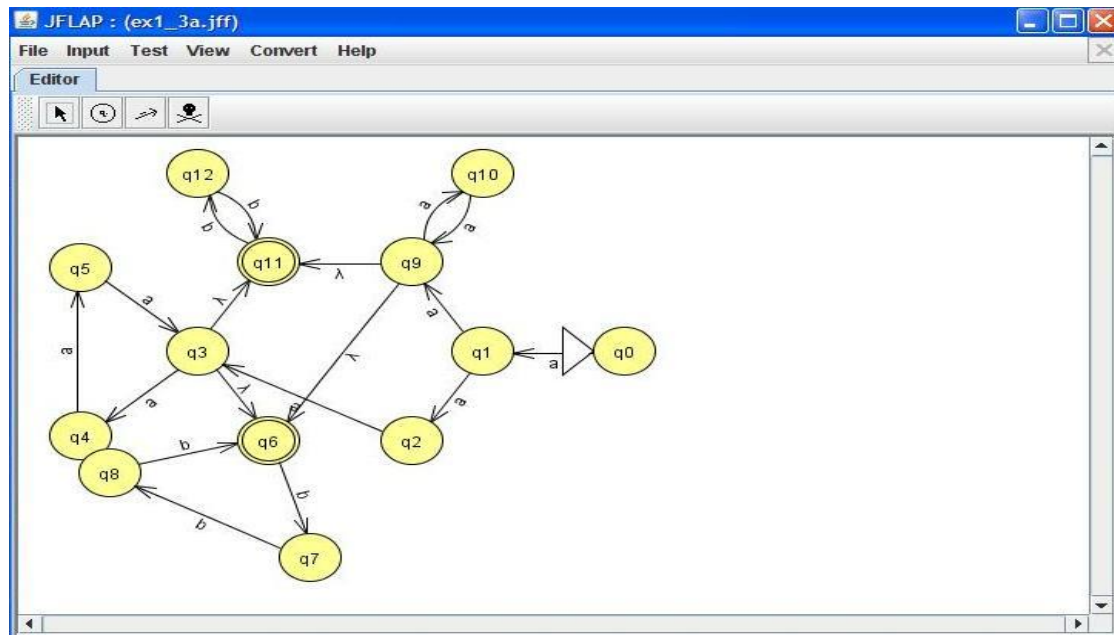
## Layout Commands (as of JFLAP version 6.2)

JFLAP will now let you apply predefined graph layout commands to your graph, which can help with a more aesthetically pleasing graph. There is a new menu in the automaton editor window, the “View” menu, which will allow one to both save the current graph layout and to apply different graph layout commands and algorithms. For a full tutorial on how to use these features, and to see a description of the built-in layout commands, feel free to read the [layout command tutorial](#).

The following are pictures of the finite automaton used earlier, [ex1.3a.jff](#), with new graph layouts. The first picture demonstrates the automaton under a “GEM” layout algorithm, the

second under a “Tree (Degree, Vertical)” layout algorithm, and the last under a “Two Circle” layout algorithm.







## 4. Converting a NFA to a DFA

### Contents

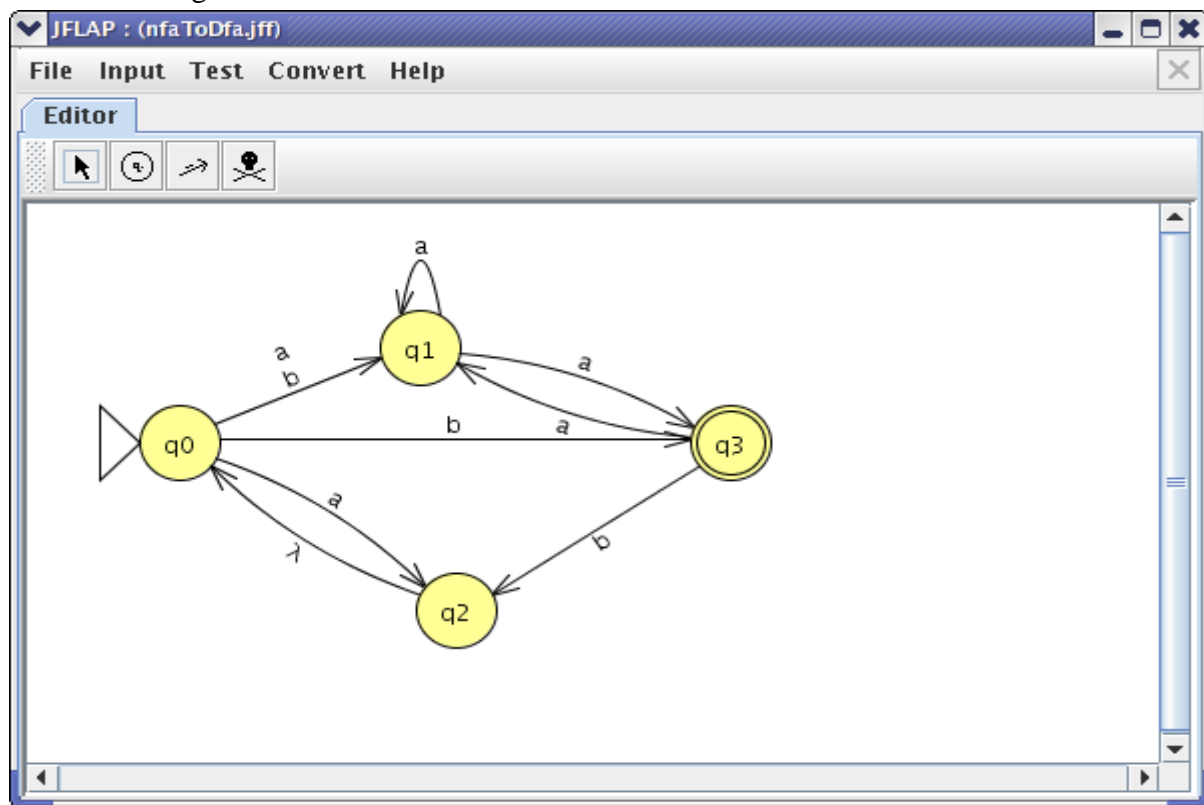
#### Introduction

#### Converting to a DFA

### Introduction

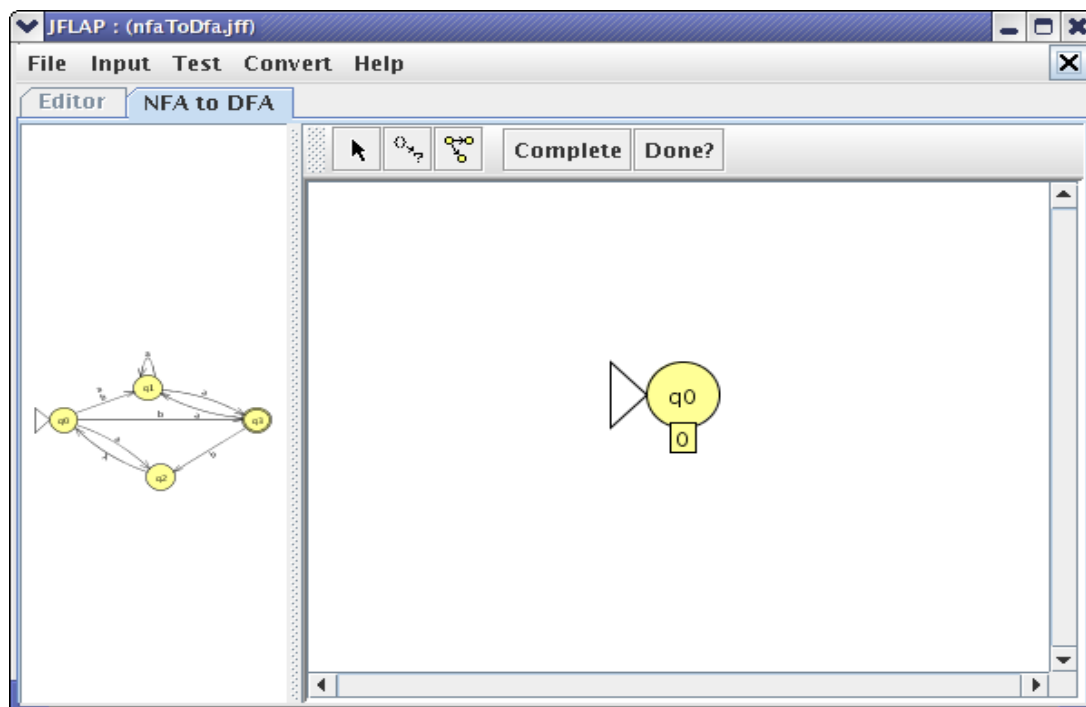
It is recommended, if you haven't already, to read the tutorial about creating a finite automaton, which covers the basics of constructing a FA. This section specifically describes how one may transform any nondeterministic finite automaton (NFA) into a deterministic automaton (DFA) by using the tools under the “Convert → Convert to DFA” menu option.

To get started, open JFLAP. Then, either load the file nfaToDfa.jff, or construct the nondeterministic finite automaton present in the screen below. When finished, your screen should look something like this:

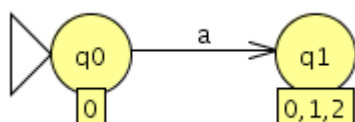


### Converting to a DFA

We will now convert this NFA into a DFA. Click on the “Convert → Convert to DFA” menu option, and this screen should come up:



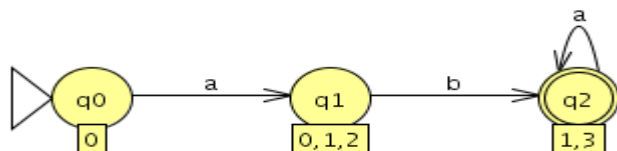
The NFA is present in the panel on the left, and our new DFA is present in the screen to the right. Let's begin building our DFA. First, hold the mouse over the second button in the toolbar from the left to see the description of the button, which should be "Expand Group on (T)erminal". Click on that button, and then click and hold down the mouse on state q0. Then, drag the mouse away from the state, and release it somewhere in the white part of the right panel. Enter "a" when you are prompted for the terminal, and "0,1,2" when prompted for the group of NFA states. When finished, your screen should contain something like this (Note: one may adjust the states so that the layout resembles this image):



You are probably curious about what exactly we just did. Basically, we just built the first transition in our DFA. In our NFA, we start at state "q0". This is represented in the DFA through the "0" label in DFA state "q0". The DFA's "a" expansion represents what happens when processing an "a" in the NFA. However, in the NFA there are three possible states we can proceed to when processing an "a", NFA states "q0" and "q1", and "q2". This multiplicity of possibilities is represented by the label under DFA state "q1", which is "0,1,2" (standing for NFA states "q0", "q1", and "q2").

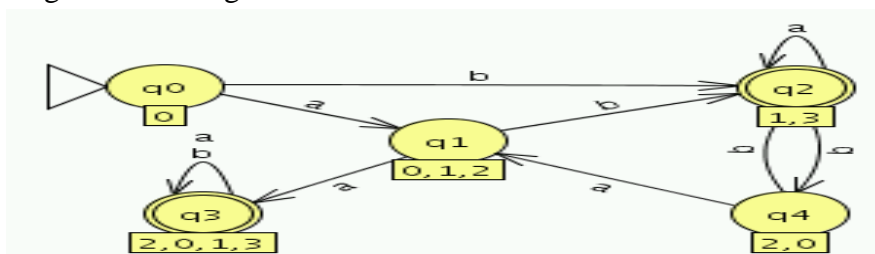
You should note that it is not possible to expand a state on a terminal that does not have a corresponding path in the NFA. For example, if one tries the same method above on state "q0", but instead uses the terminal "c", an error message will pop up.

Now we can add two more transitions in the DFA. Through the same method as before, add another expansion, starting from "q1". The terminal value will be "b" and the states will be "1,3". This will create a new state "q2". Now, add an expansion starting from "q2". This time, however, release the mouse on "q2" and enter the terminal "a". When finished, something resembling the following should be present on your screen:



From NFA state “q1”, there are no expansions leading from it that contain a “b”. From NFA states “q0” and “q2”, a “b” will allow the program to reach NFA states “q1” or “q3”. Thus, the label for DFA state “q2” is “1,3”. You also might notice that DFA state “q2” is a final state. Since NFA state “q3” is a final state, and DFA state “q2” represents a possible path to NFA state “q3”, DFA state “q2” is a final state. It does not matter that DFA state “q2” also represents a possible path to NFA state “q1”, a nonfinal state. As long as one possible path represented by a DFA state is final, the DFA state is final. In the case of the expansion from “q2” to “q2”, since in DFA state “q2” one can be at NFA states “q1” or “q3”, and since processing any “a” from there will result in being in one of those two NFA states, the expansion does not result in a DFA state change.

Let's check and see if we are done. Go ahead and click on the “Done?” button. The message will tell us that we still have some work to do, as we have two more states and seven more transitions unaccounted for in the DFA. Go ahead and finish it. One can utilize, if desired, two other buttons in the toolbar. The “(S)tate Expander” button, if pressed, will fill out all transitions and create any new DFA states those transitions require when you click on an existing DFA state. Alternatively, if one wants to see the whole DFA right away, one can click on the “Complete” button. If either of these are pressed, the layout manager may not put everything onto the part of the screen currently visible, so one may have to maximize the window or find all states using the sliders. After finishing the DFA, and perhaps after moving the states around a little, you should have a picture resembling the following:



Now click the “Done?” button again. After a message informing you that the DFA is fully built, a new editor window is generated with the DFA in it. Congratulations, you have converted your NFA into a DFA!

## 5. Converting a DFA to a Minimal State DFA

### Contents

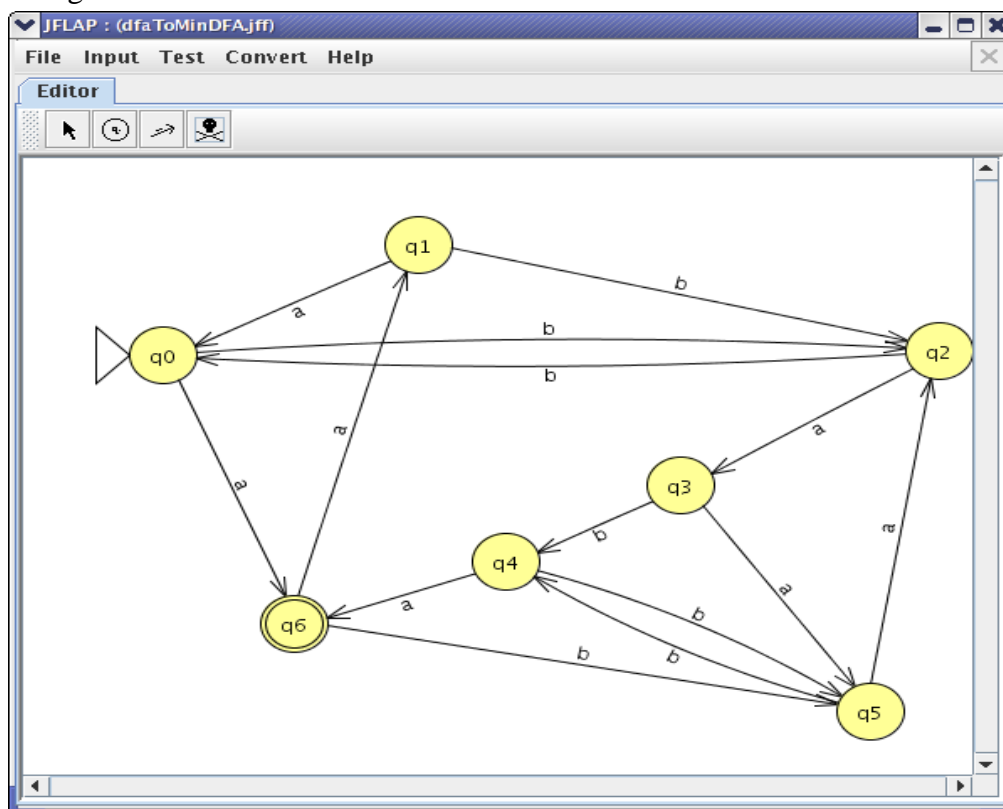
#### Introduction

#### Converting to a Minimal DFA

### Introduction

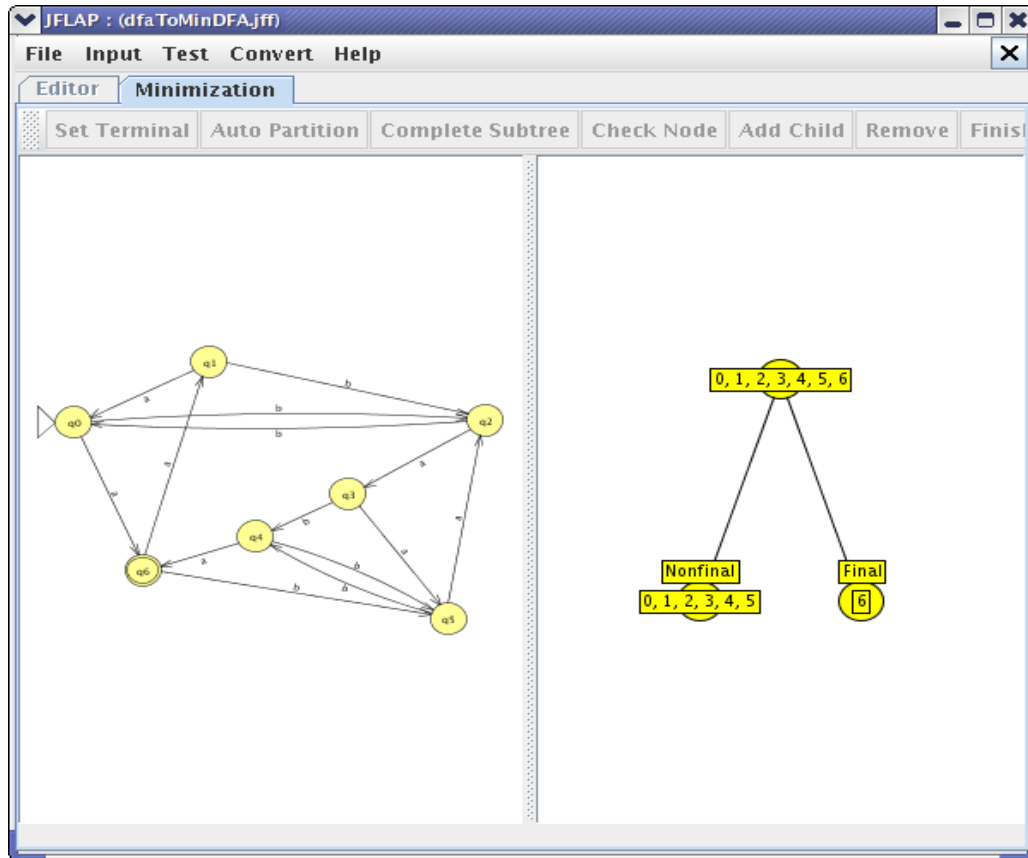
It is recommended, if you haven't already, to read the tutorial about [creating a finite automaton](#), which covers the basics of constructing a FA. This section specifically describes how one may transform any deterministic finite automaton (DFA) into a minimal state deterministic finite automaton (a DFA with a minimal number of states) by using the tools under the “Convert → Minimize DFA” menu option.

To get started, open JFLAP. Then, either load the file [dfaToMinDFA.jff](#), or construct the nondeterministic finite automaton present in the screen below. You may have to resize your screen a little if the whole graph does not fit onto the screen. When finished, your screen should look something like this:



### Converting to a Minimal DFA

We will now convert this DFA into a minimal state DFA. Click on the “Convert → Minimize DFA” menu option, and this screen should come up:



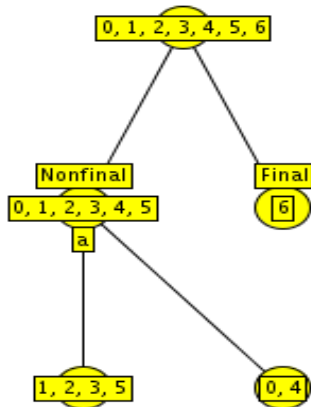
The DFA is present in the panel on the left, and a tree of states is present in the screen to the right. You might wonder what exactly the tree nodes and labels in the right panel mean. At present, in the root node, all states in the original DFA are represented. In the tree we group all the states from the DFA together in the root of the tree. Then examining these states we look for ways to distinguish them. If we can distinguish states in a node, then we split a node into two or more nodes of states that are distinguished from each other. One way to do this is to divide states in a leaf node and assign them to new children of the node according to their outcomes when grouped along a terminal. We can repeat this until there are no more splits. At that time, the leaf nodes represent the states in the minimal DFA. If a leaf node has multiple states from the original DFA, these states can be combined in the minimal DFA.

Before splitting along terminals, we first split the nodes into "Final" and "Nonfinal" groups. If a state in the DFA is a final state, it is represented in the "Final" node in the minimal DFA as part of the "Final" node's label. As shown, only "q6" in the DFA is a final node, so only a "6" is in the "Final" node label. The same can be applied for nonfinal DFA states and the "Nonfinal" node. The "Nonfinal" node has the values "0, 1, 2, 3, 4, 5" because all other DFA states are nonfinal.

Let's begin splitting the leaf nodes in the tree of states. First, click on the "Nonfinal" node. You will probably notice that all the DFA states represented by the "Nonfinal" node are highlighted to the left and that some buttons are now visible. Click on one of the newly visible buttons, "Set Terminal". A pop-up will appear prompting for a terminal. Enter the letter "b". Instead of something happening to the node, a warning message will appear stating that the group doesn't split on that terminal. This is because there is no transition containing a "b" pointing from a member of the "Nonfinal" group to a member of any other group. All transitions that contain a "b" either change between states within the group or else point into the group from outside it.

Now, try to expand on the terminal "a". When this happens, one will see two child nodes issuing from the "Nonfinal" node. Our task now is to divide the nodes present in "Nonfinal". We will

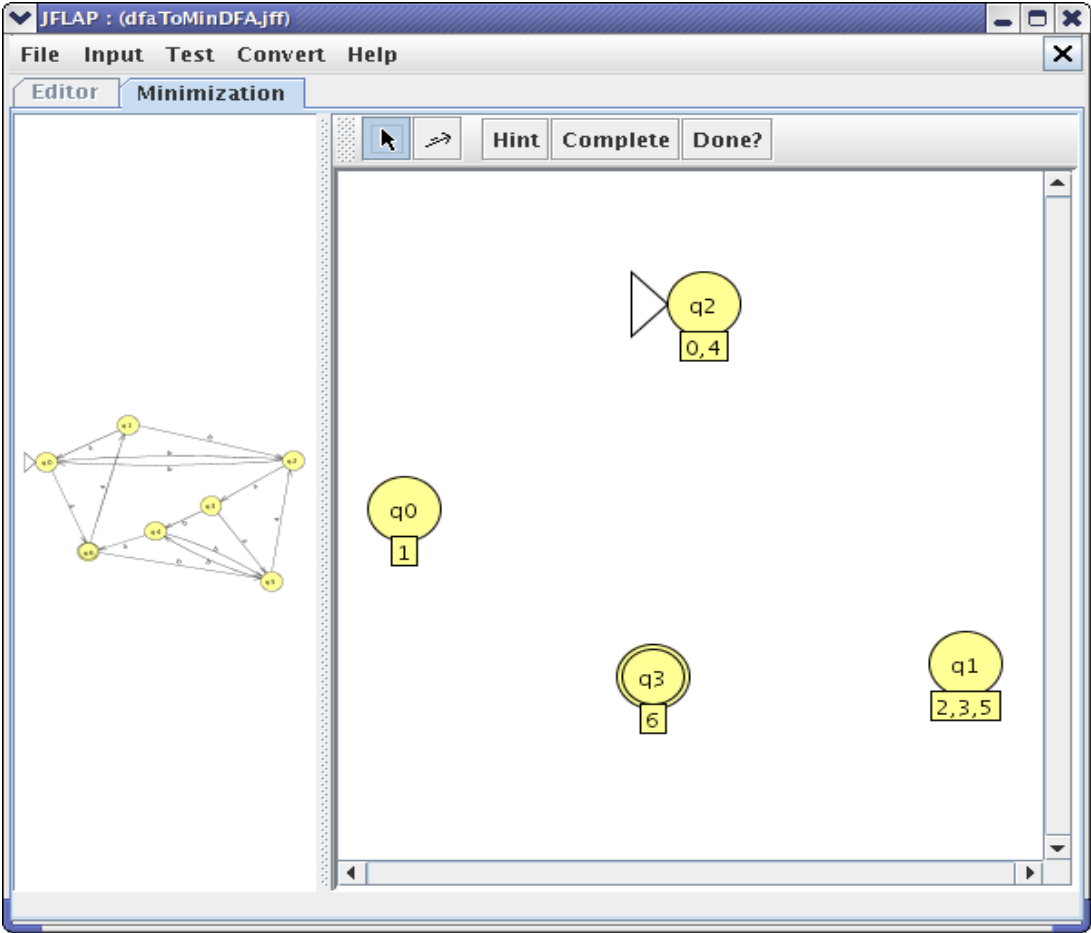
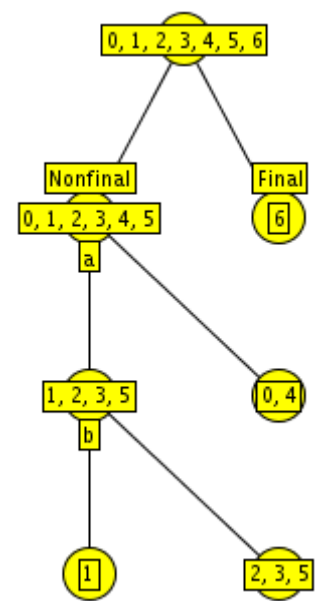
move to one child all the states which on an “a” terminal end up in state 6 (one group of states in a leaf node), and move to the other child all the states which on an “a” terminal end up in states 0-5 (the other group of states in a leaf node). States “q0” and “q4” both transition along the terminal “a” to “q6”, the only node not in the “Nonfinal” group. Thus, “q0” and “q4” need to be in one child. Click on the right child, and then click on “q0” and “q4” in the left panel. Next, click on the left child and then click on the remaining “Nonfinal” states in the left panel. When finished, press the “Check Node” button to receive a message stating whether the expansion is correct. If so, the tree should look like this...



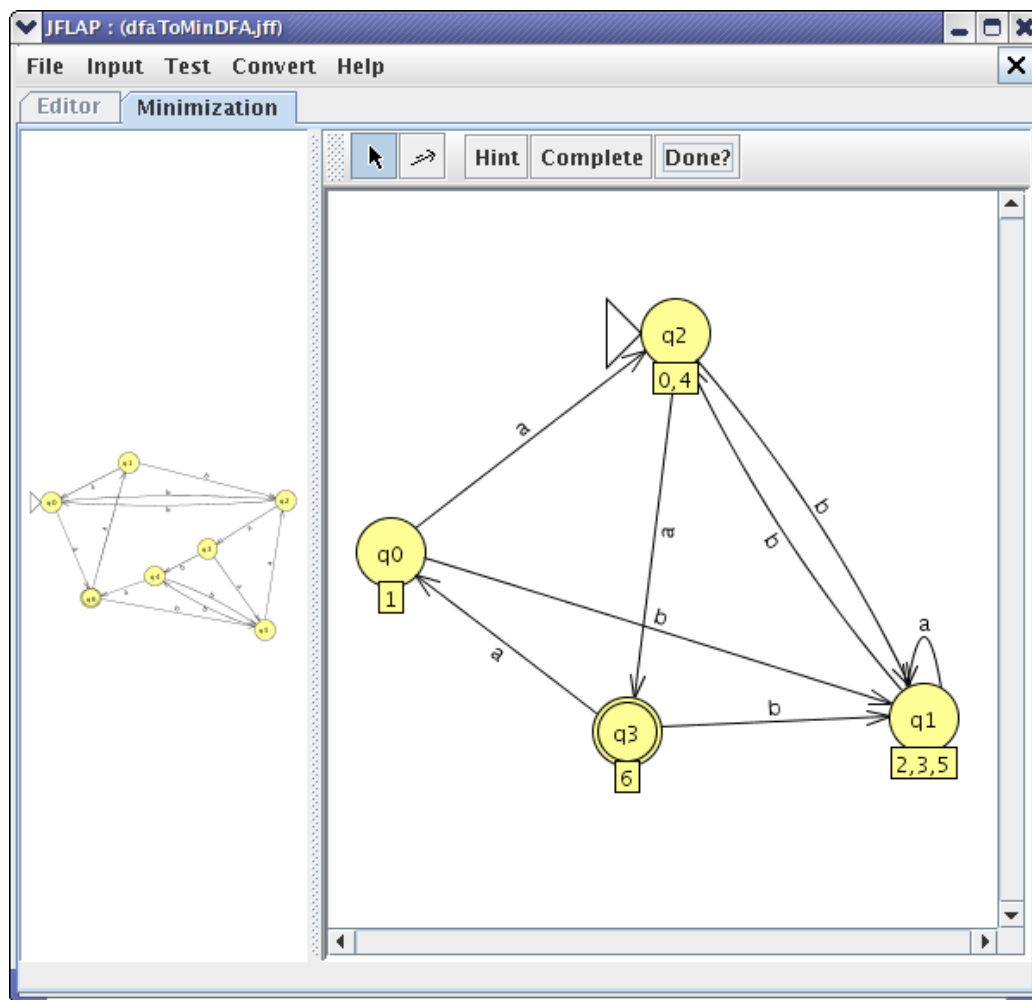
Now, let's finish the minimal DFA. Click on the left child node, and select the “Auto Partition” or “Complete Subtree” buttons. The difference between the two buttons is that “Auto Partition” will simply figure out a division for the currently selected child, while “Complete Subtree” will complete all divisions for which the selected node is an ancestor. Whichever one is clicked, the nodes in the leftmost leaf will now divide according to the terminal “b” (although it might also divide according to terminal “a”). When done, click the newly visible “Finish” button, and the states in the minimal DFA will be shown (note: the states may be in different places on the screen than in the caption below).

Complete tree

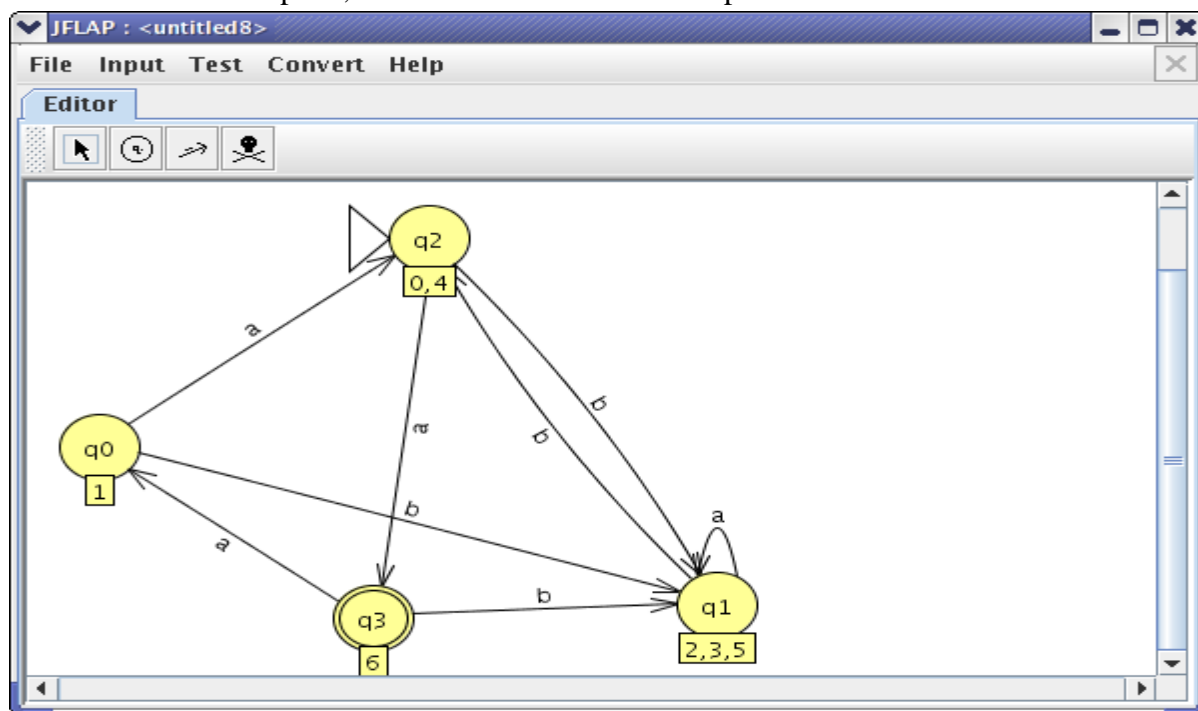
New minimal states



We now need to add the transitions. Let's have our first one be from “q2” to “q3” along the terminal “a”, which will represent the transitions from both “q0” and “q4” to “q6” along “a” in the original DFA. Click on the transition button, the second one from the left in the button toolbar, and create the transition. Once done, go ahead and add the rest of the transitions. If you get stuck, feel free to utilize the “Hint”, “Complete”, or “Done?” buttons. The “Hint” button will fill in one transition that needs to be completed, the “Complete” button will fill in all remaining transitions, and the “Done?” button will inform you of how many transitions you have remaining. Note also that if you try to create a transition that doesn't exist in the original DFA, such as between “q2” and “q3” along terminal “b”, then an error message will come up. When finished, your picture should resemble the one below:



Now, to export the file, click on the “Done?” button. After seeing a message stating that the minimal DFA is complete, the minimal DFA should be present in a new window.





## 6. Converting a DFA to a Regular Grammar

### Contents

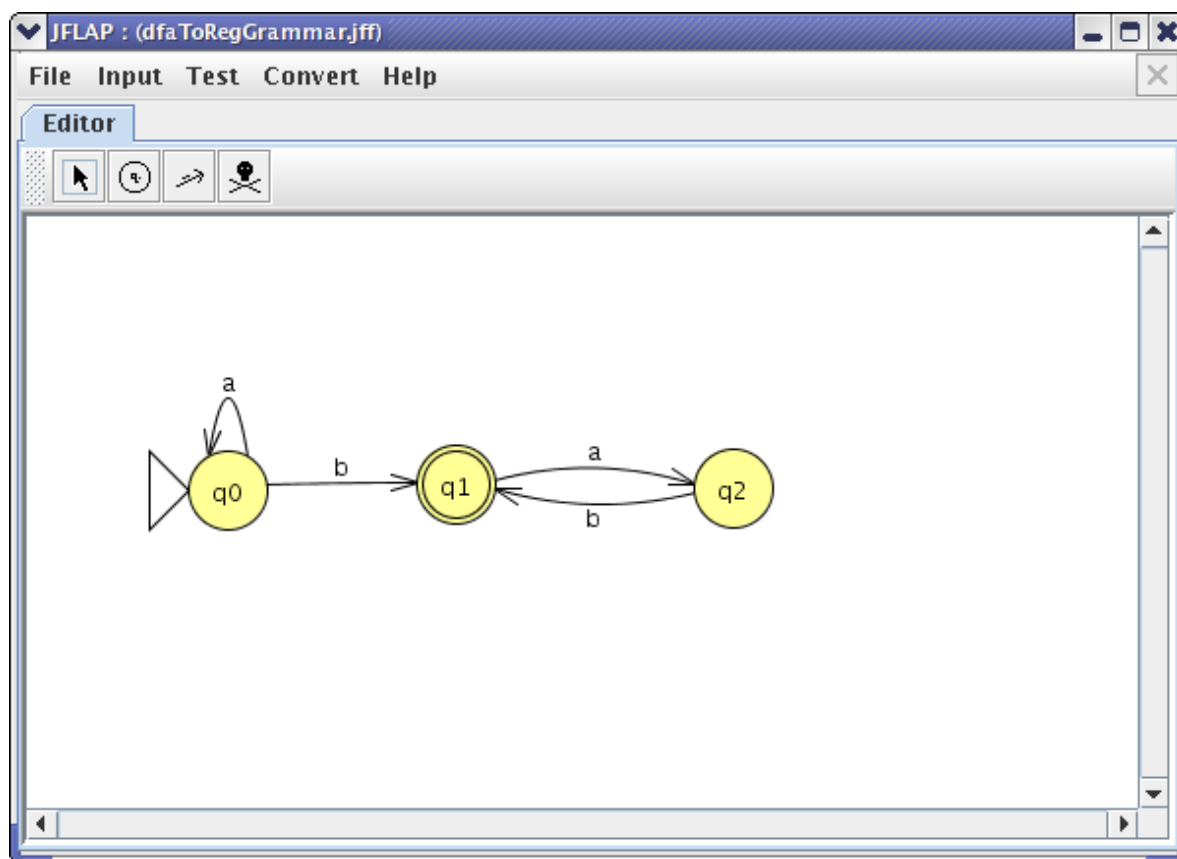
#### Introduction

#### Converting to a Regular Grammar

### Introduction

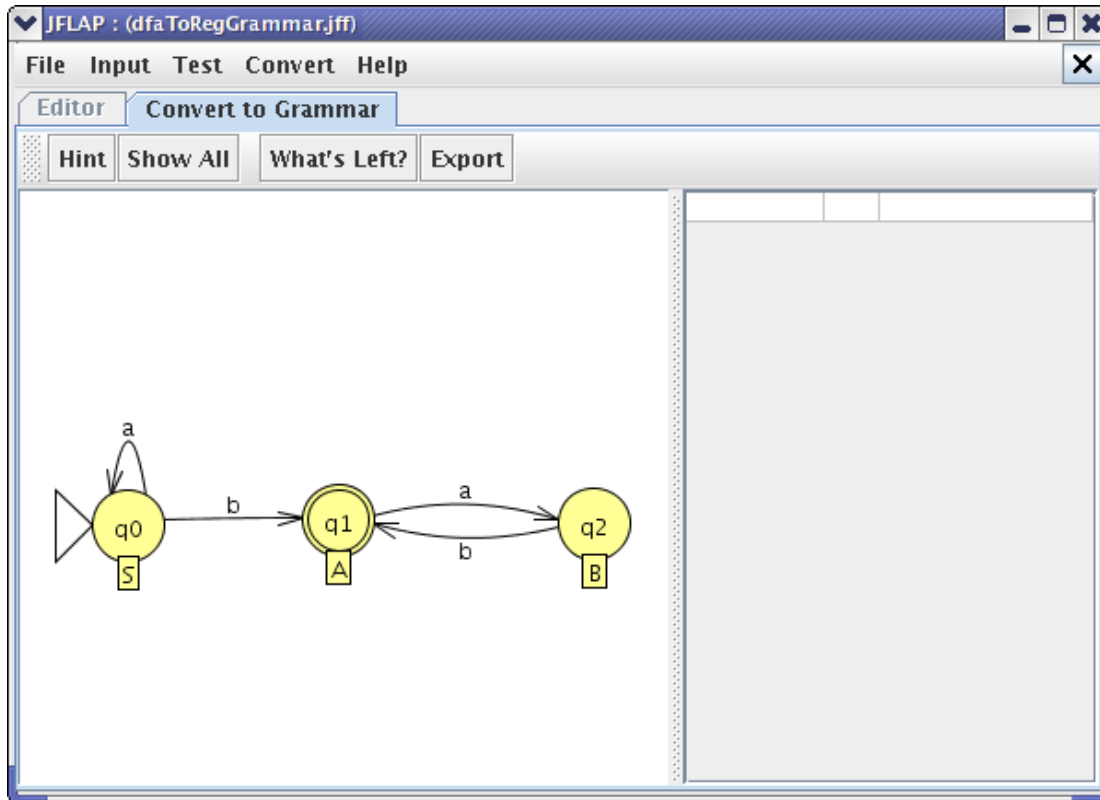
The reader, if he or she hasn't already, should read the tutorial about creating a finite automaton, which covers the basics of constructing a FA and describes how they are implemented in JFLAP. This section specifically describes how one may transform one of these finite automata into a right-linear grammar.

To get started, open JFLAP. Then, either load the file dfaToRegGrammar.jff, or construct the nondeterministic finite automaton present in the screen below. When finished, your screen should look something like this:



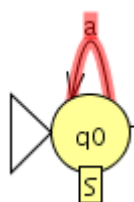
### Converting to a Regular Grammar

We will now convert this DFA into a regular grammar. Click on the "Convert → Convert to Grammar" menu option, and this screen should come up:



Notice the labels on the states in the picture. JFLAP, by default, assigns unique variables to each state. Each transition in the graph corresponds with a production in a right-linear grammar. For example, the transition from “q0” to “q0” along the transition “a” matches the production “ $S \rightarrow aS$ ”. Let's add this production to our list of productions in the right panel. To add the production, just click on the “a” transition on state “q0”. The “a” transition should now be highlighted in red, and the production added to the right panel.

### Highlighted Transition   Corresponding Production

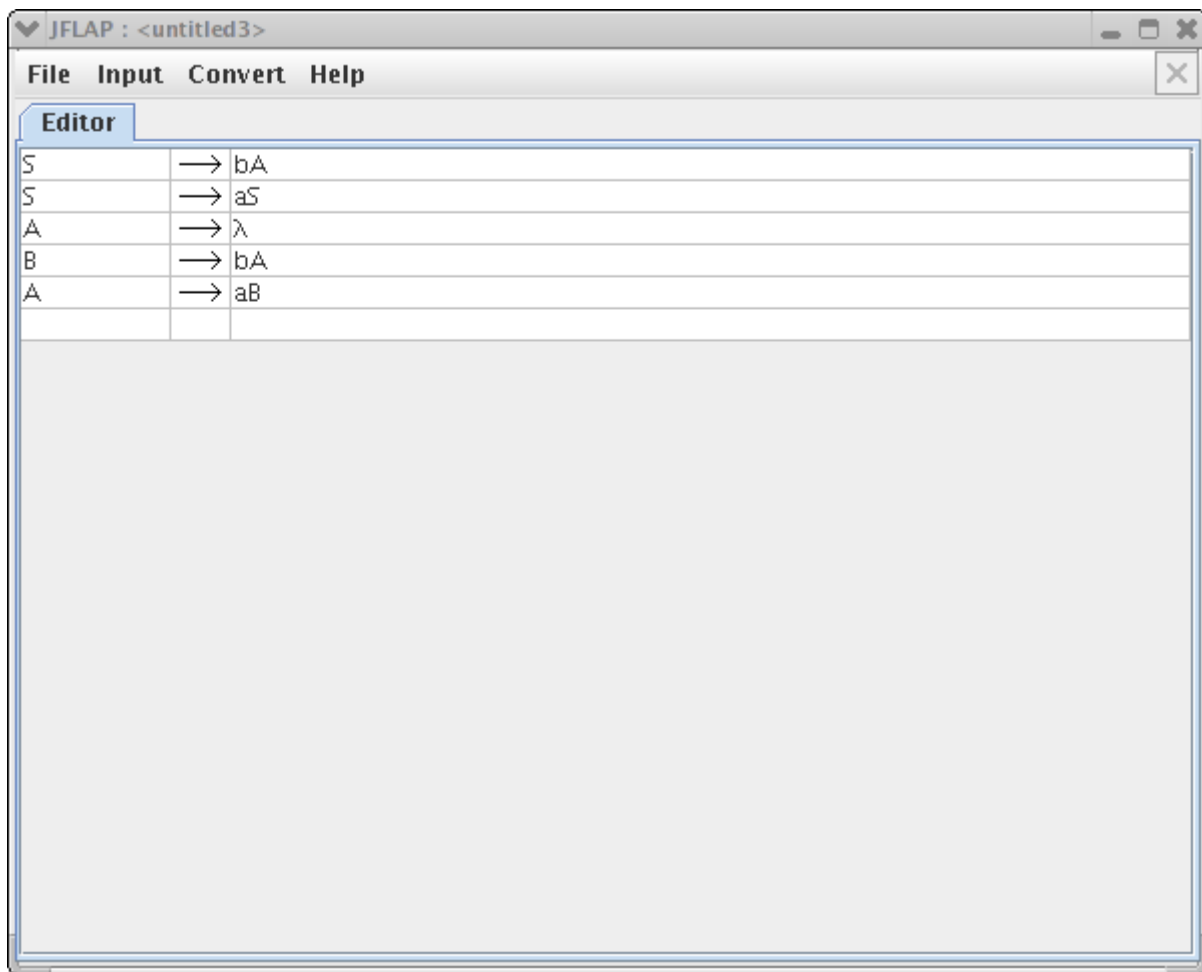


S	→	aS

Now, let's see what our next step might be. We know it probably will be a transition in the FA, but if we are having trouble, we can click on the “Hint” button, and one production that we haven't generated yet will be generated. Now, how many productions do we still need to generate. Click on the “What's Left?” button to find out. We will see the remaining two transitions highlighted, but we will also see state “q1” highlighted. This is because any final state carries the production to  $\lambda$ . Feel free to click on the rest of transitions and the final state, or the “Show All” button, which will add the rest of the productions to the graph. After finishing, the following should be your list of productions (even if not exactly in that order):

B	→	bA
S	→	bA
A	→	$\lambda$
A	→	aB
S	→	aS

When finished, click on the “Export” button, and you should have a new grammar on the screen resembling the screen below:



## 7. Converting a FA to a Regular Expression

### Contents

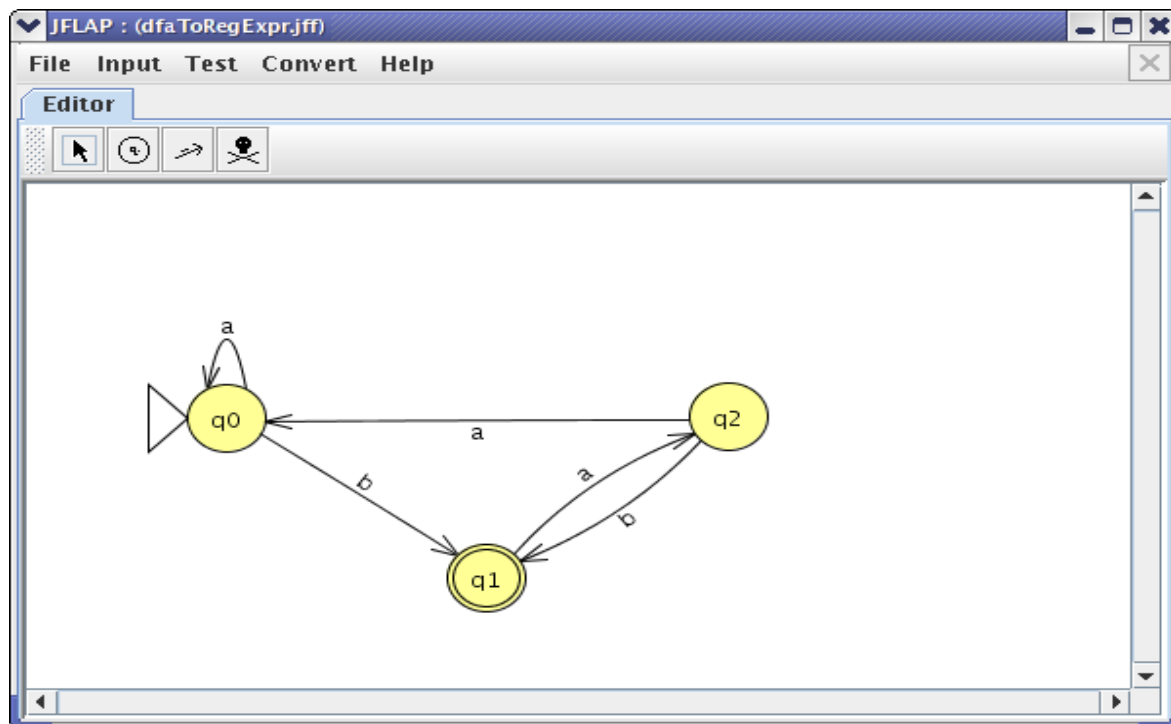
#### Introduction

#### Converting to a Regular Expression

### Introduction

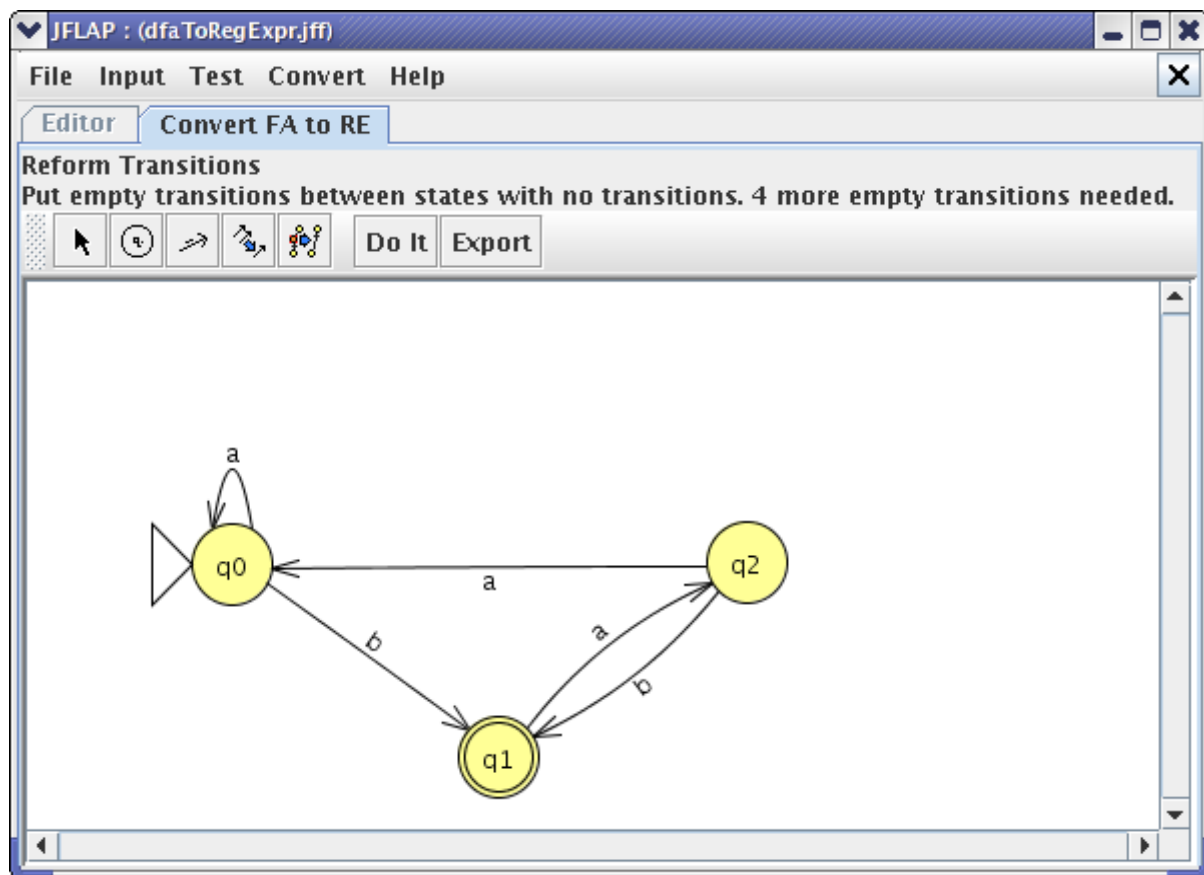
This section specifically describes how one may transform any finite automaton into a regular expression by using the tools under the “Convert → Convert FA to RE” menu option. For knowledge of many of the general tools, menus, and windows used to create an automaton, one should first read the tutorial on finite automata. For knowledge of regular expressions and how JFLAP defines and implements them, one should first read the tutorial on regular expressions.

To get started, open JFLAP. Then, either load the file dfaToRegExpr.jff, or construct the finite automaton present in the screen below. When finished, your screen should look something like this:



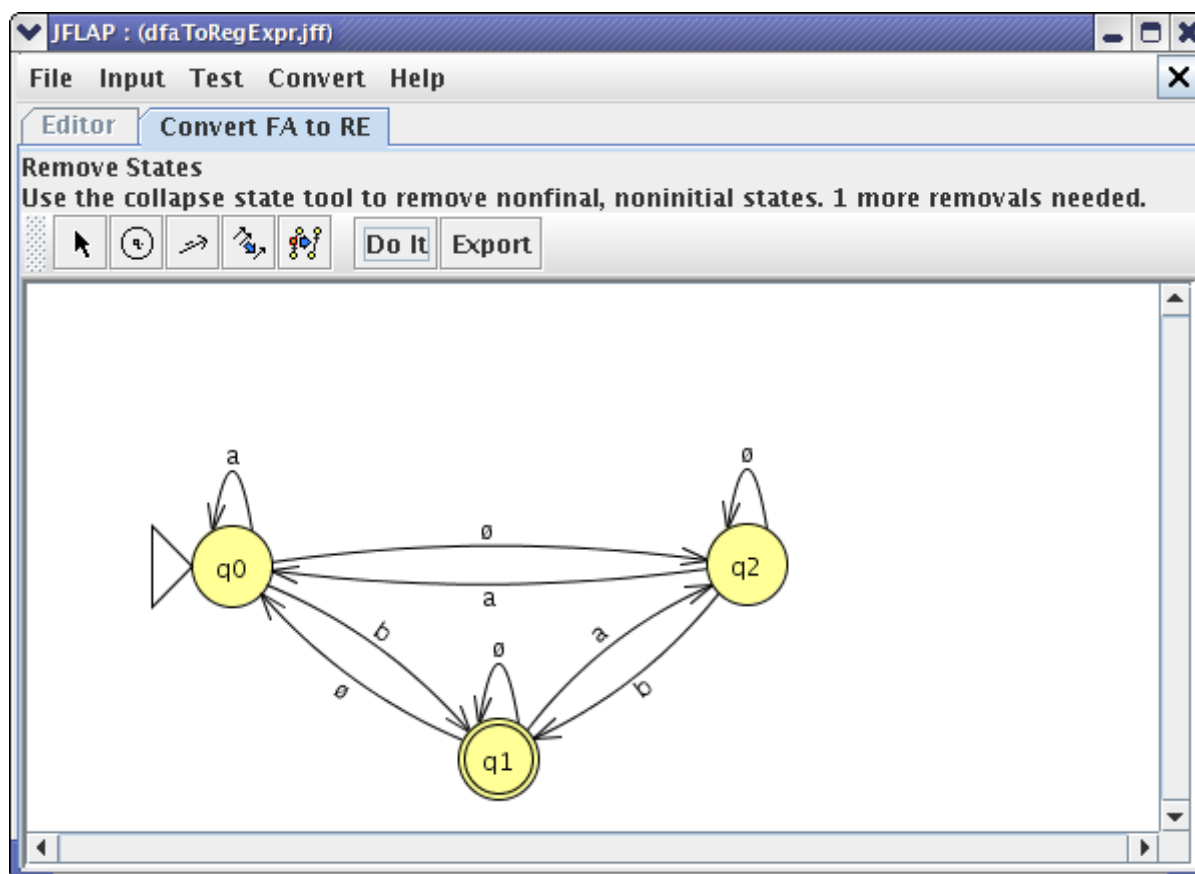
### Converting to a Regular Expression

We will now convert this DFA into a regular expression. Click on the “Convert → Convert FA to RE” menu option, and this screen should come up:

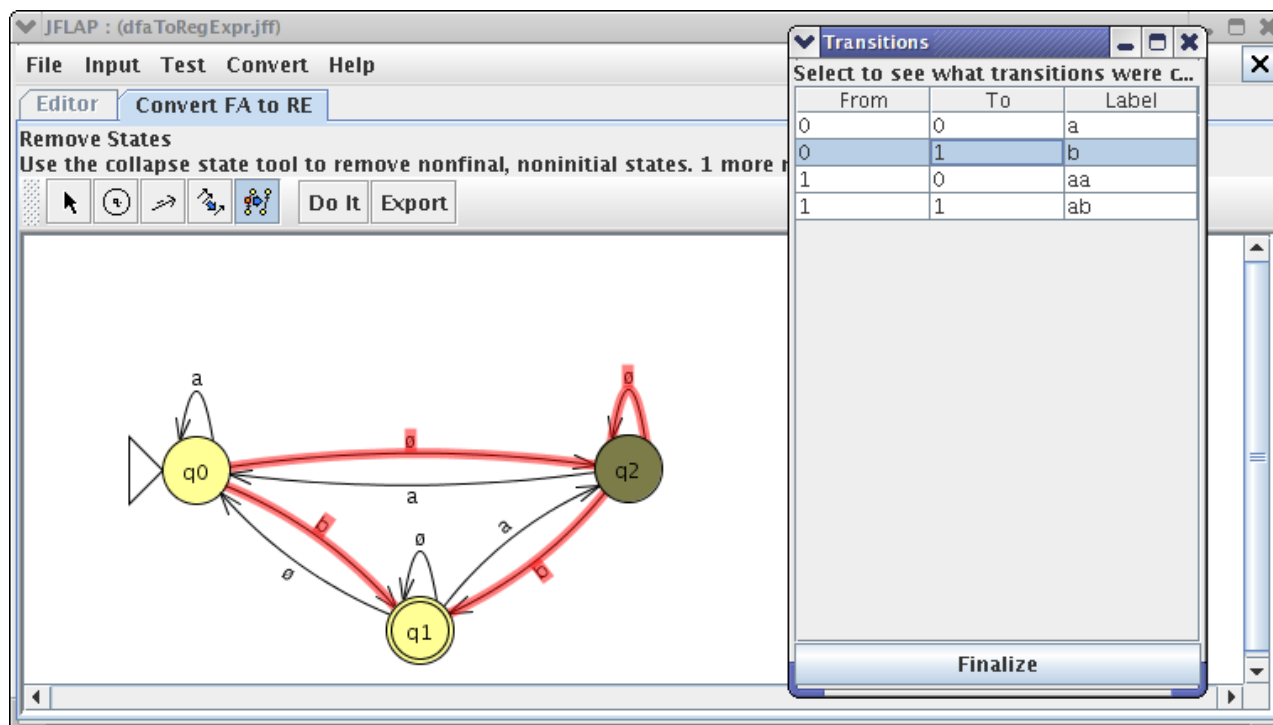


You can see that this is a fairly simple automaton, and that we are prompted for 4 new transitions. If this were a more complex automaton, with multiple final states, we would first have to create a single final state and then establish “ $\lambda$ ” transitions between the old final states and the new final state. Also, if there was more than one transition between two states, we would have to use the “Transition (C)ollapser” button (fourth from left) to transform multiple transitions into one transition. This is done simply by clicking on the transition, and multiple transitions will be linked by union operators inside one transition. For example, multiple transitions of “a” and “b” will become the solo transition “a+b”. Doing these steps is fairly straightforward, so for the sake of simplicity we will not use such an example utilizing them.

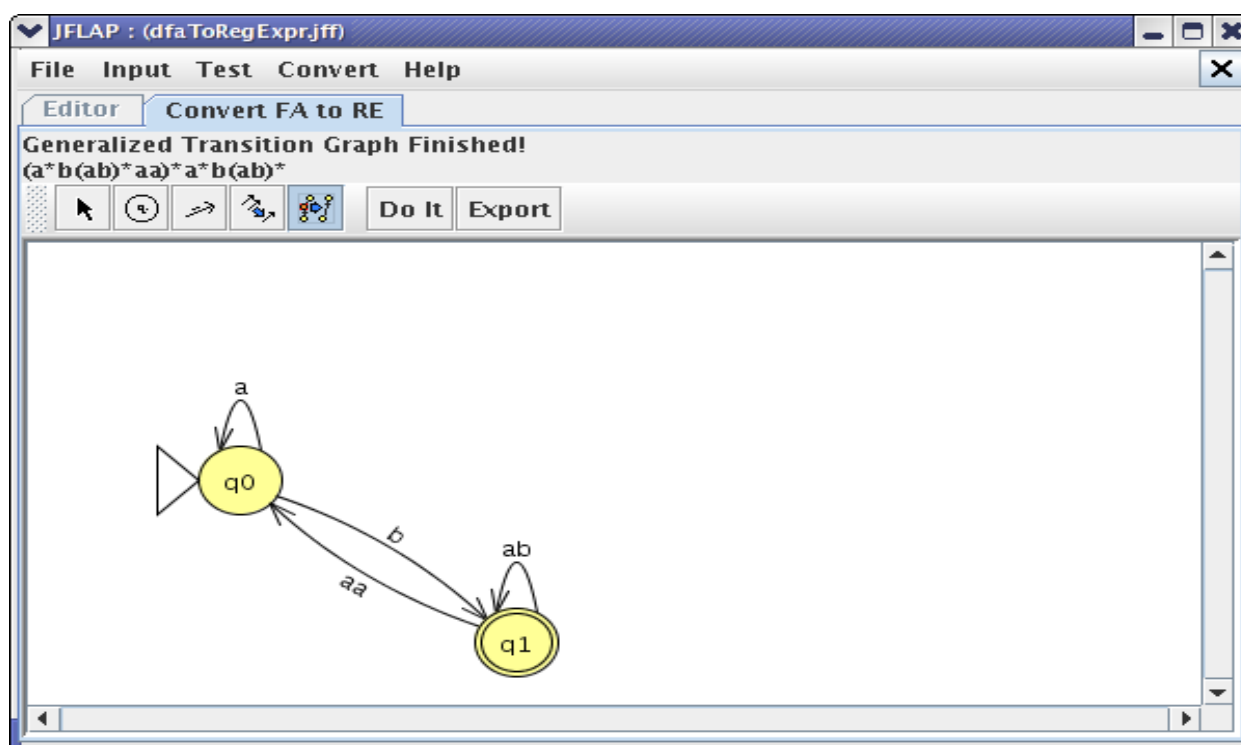
What exactly are the 4 transitions that we need to add, and why do we need to add them? Before the conversion algorithm can work, every state must have a transition to every state in the graph, including itself. Thus, the total number of transitions for a finite automaton with  $n$  states must be  $n^2$ . In this example, since there are 3 states, there should be 9 transitions. Since, however, the traditions that aren't currently defined are never taken, the transitions that we add will always be on the empty set of strings, or  $\emptyset$ . Now, let's add the 4 transitions. Add transitions from “q1” to “q1”, “q2” to “q2”, “q2” to “q0”, and from “q1” to “q0” by first clicking on the “(T)ransition Creator” button (third from left) and then on the appropriate places on the screen. When finished, the screen should resemble the one below...



Now the message has changed, and it tells us to use the collapse state tool. This tool is accessed using the fourth button from the left, the “State (C)ollapser” button. Click on that and then try to click on either state “q0” or “q1”. You should be informed that the final or initial state cannot be removed (whichever one you clicked on). Thus, the only state that can be removed is “q2”. Thus, click on that state. Once you click on it, a “Transitions” window should come up, which lists all the transitions and labels that will be present in the current graph once the current state has been removed. If you click on the individual rows in the window, the old transitions in the editor window that correspond to the new transition will be highlighted, in addition to the state that will be removed. The screen below is an example of both the Transition window, highlighted old transitions, and the highlighted state to be removed (in a resized main window).



Click on “Finalize” and the “Transitions” window will disappear. The following screen should now be visible, with the new regular expression under the “Generalized Transition Graph Finished!” label. Note that since the generalized transition graph has only 2 states, it is easy to figure out the regular expression. It consists of all cycles from “q0” to “q0”  $[(a^*b(ab)^*aa)^*]$ , followed by the loop transition on “q0” zero or more times  $[a^*]$ , followed by the transition from “q0” to “q1”  $[b]$ , and concluding with the loop transition on “q1” zero or more times  $[(ab)^*]$ . Concatenating them together, you will have the regular expression  $[(a^*b(ab)^*aa)^*a^*b(ab)^*]$ . Click on the “Export” button, and you will now have this new regular expression to use as you see fit.



## 8. Mealy Machines

### Contents

#### Definition

#### Differences between a Mealy Machine and an FA

- No Final State
- Transition Output
- Producing Output from an Input String
- No Nondeterminism

#### Proceed to Mealy Machine Examples

### Definition

JFLAP defines a Mealy machine  $M$  as the sextuple  $M = (Q, \Sigma, \Gamma, \delta, \omega, q_s)$  where

$Q$  is a finite set of states  $\{q_i \mid i \text{ is a nonnegative integer}\}$

$\Sigma$  is the finite input alphabet

$\Gamma$  is the finite output alphabet

$\delta$  is the transition function,  $\delta : Q \times \Sigma \rightarrow Q$

$\omega$  denotes the output function,  $\omega : Q \times \Sigma \rightarrow \Gamma$

$q_s$  (is a member of  $Q$ ) is the initial state

Mealy machines are different than Moore machines in the output function,  $\omega$ . In a Mealy machine, output is produced by its transitions, while in a Moore machine, output is produced by its states.

To start a new Mealy machine, select the **Mealy Machine** option from the main menu.






## Differences between a Mealy Machine and an FA

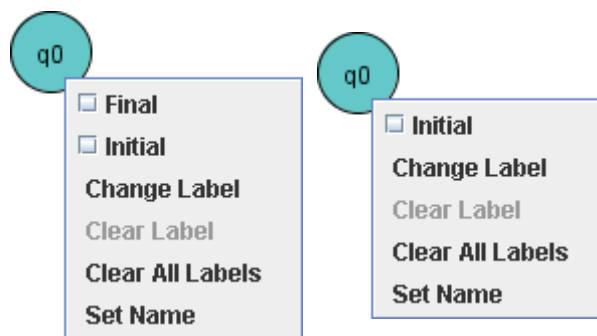
A Mealy machine is very similar to a Finite Automaton (FA), with a few key differences:

- It has no final states.
- Its transitions produce output.
- It does not accept or reject input, instead, it generates output from input.
- Lastly, Mealy machines cannot have nondeterministic states.

Let's go through these points.

### No Final State

In an FA, when you have the Attribute Editor tool selected (you may do so by clicking the  button), right-clicking on a state it will produce a pop-up menu that allows you to, among other things, set it to be a final state. In a Mealy machine, that option is not available.



An FA state menu

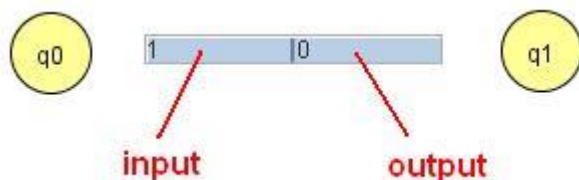
A Mealy machine state menu

A Mealy machine does not have final states because it does not accept or reject input. Instead, each transition produces output, which will be described below.

### Transition Output

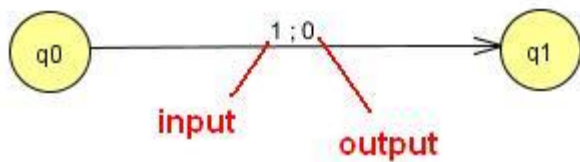
A Mealy machine produces output each time it takes a transition.

Creating a Mealy machine is the same as creating an FA with the exception of creating its transitions. In a Mealy machine, each transition produces output. When you are creating a transition, two blanks appear instead of one. The first blank is for the input symbol, the second blank is for the output symbol.



### Creating a transition

When the transition is created, its label will be two symbols separated by a semicolon, ";". The input symbol is to the left of the semicolon, and the output symbol is to its right.



Transition created

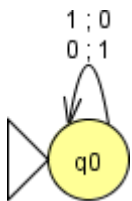
Thus, when in  $q_0$  with input of "1", the machine will take the transition to  $q_1$  and produce the output "0".

With each transition producing output, the Mealy machine can produce output from an input string.

### Producing Output from an Input String

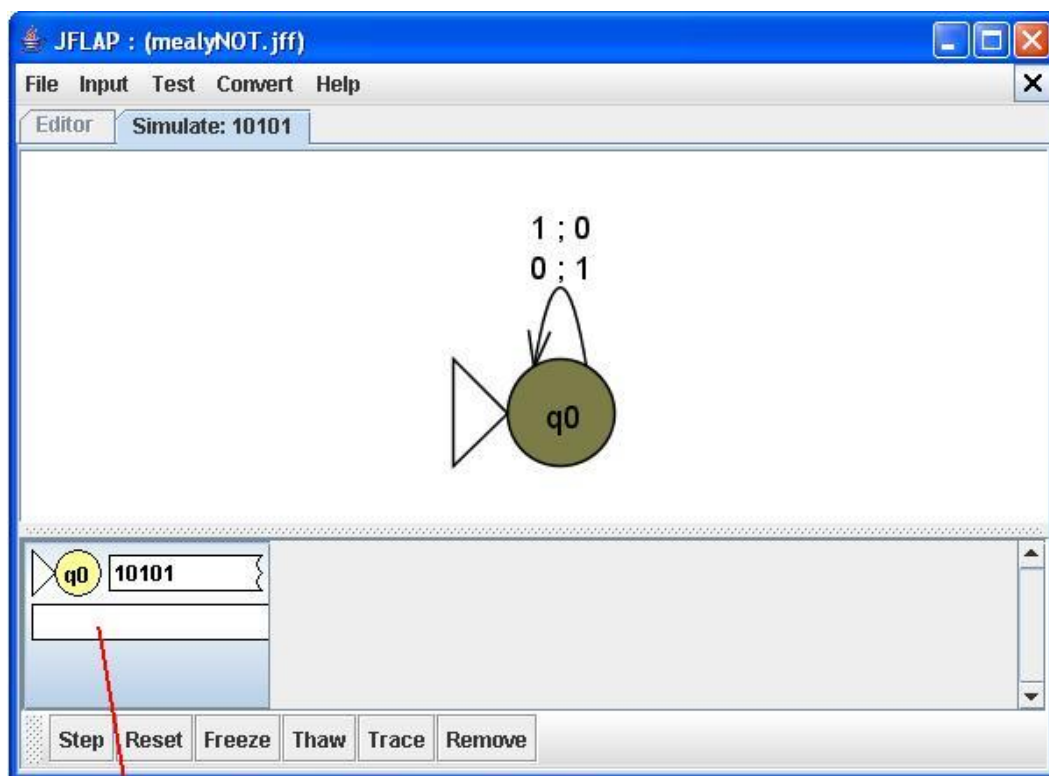
Instead of accepting or rejecting input, a Mealy machine produces output from an input string.

Let's look at this simple Mealy machine, which can be downloaded through [mealyNOT.jff](#):



A simple Mealy machine

When we select the menu option **Input : Step...** and enter your input, we get a display similar to that of an FA, except with another piece of tape below the input tape. This displays the output of the machine at the current step.



output display

The output display

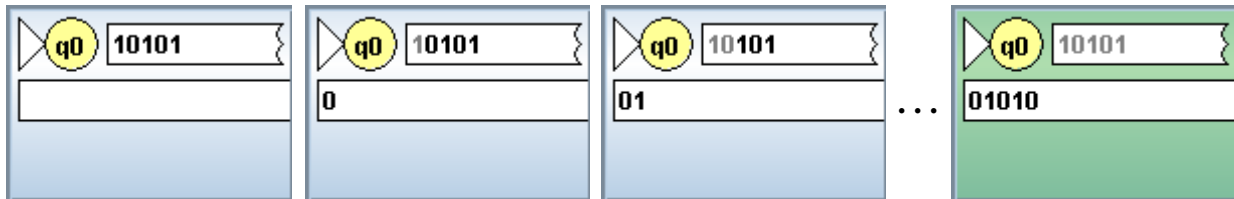
At each step. The output tape updates itself to display the total output that has been produced so far:

Initial:

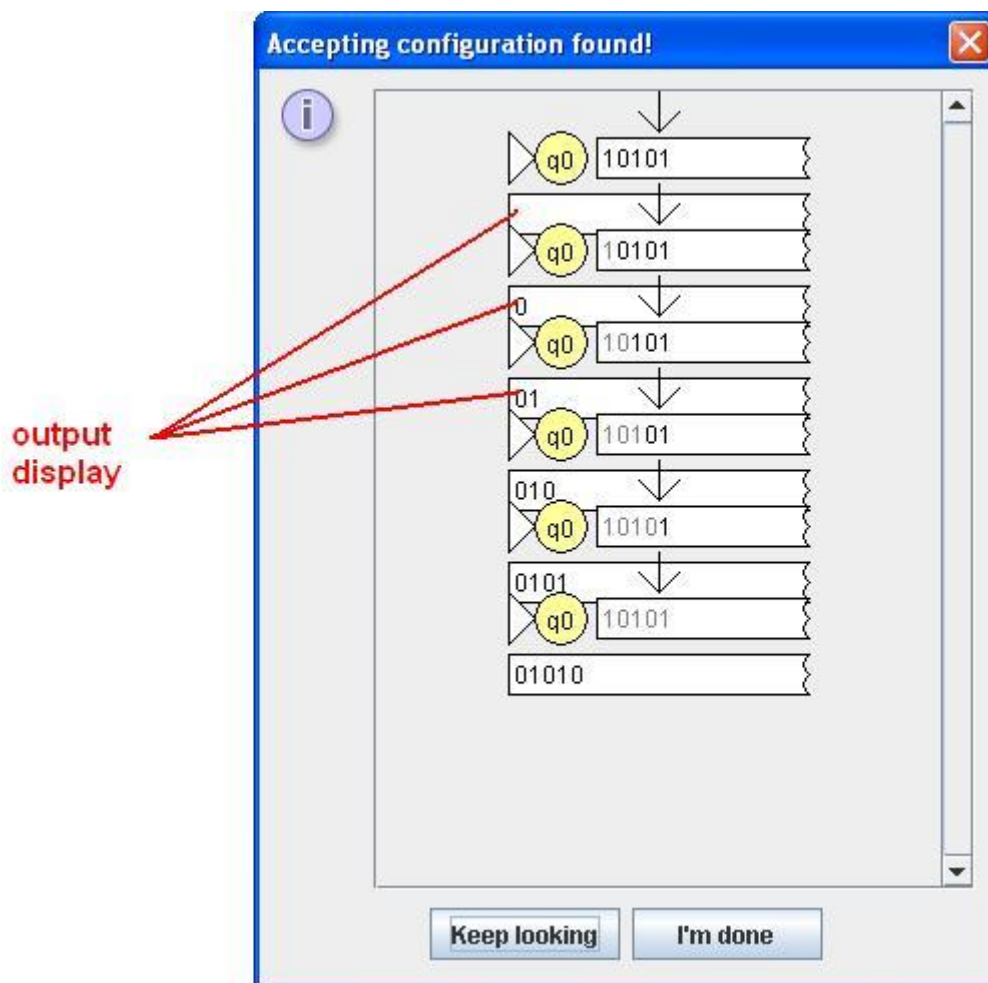
Step 1:

Step 2:

... Final:

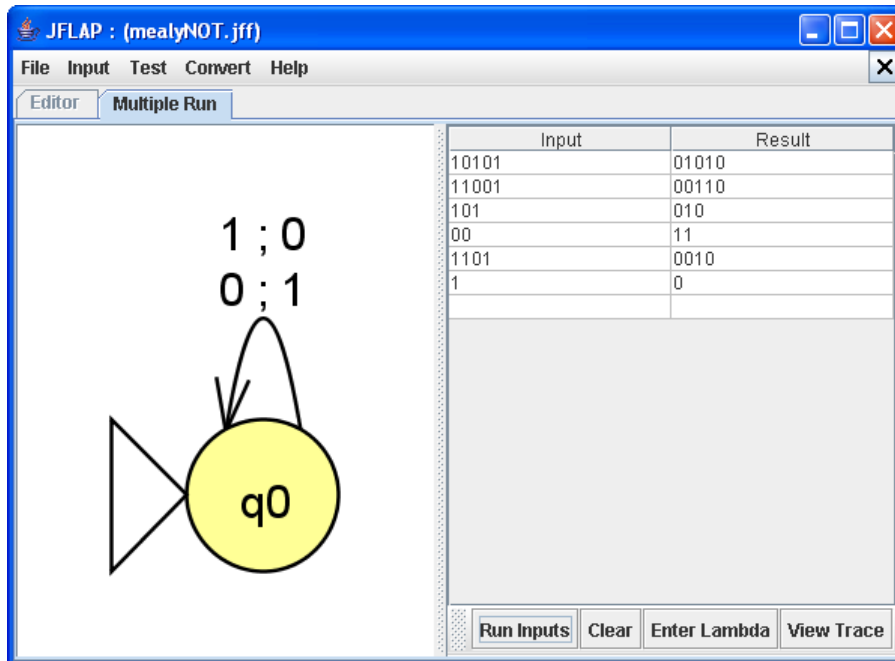


The menu option **Input : Fast Run...** works similarly, with the output being displayed in a tape under the input tape:



Fast run output display

Similarly, when we select **Input : Multiple Run**, output is displayed in the **Result** column.

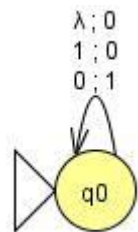


Multiple run output display

### No Nondeterminism

As Mealy machines map an input to a unique output, nondeterminism cannot exist in a Mealy machine. Although we still will be able to build a Mealy machine that has nondeterminism, we will not be able to run input on it.

For instance, let's modify our machine slightly to make:



Mealy machine with nondeterminism

We we try to run it on an input, with **Input : Step...**, **Input : Fast Run...**, or **Input : Multiple Run...**, we will get an error message asking us to remove the nondeterministic states:



Select **Test : Highlight Nondeterminism** to view the nondterministic states. Remove the nondeterminism in order to be able to run input on the Mealy machine.

## 9. Moore Machines

### Contents

#### Definition

#### Differences between a Moore Machine and an FA

- No Final State
- State Output
- Producing Output from an Input String
- No Nondeterminism

#### Output Convention

#### Proceed to Moore Machine Examples

### Definition

JFLAP defines a Moore machine  $M$  as the sextuple  $M = (Q, \Sigma, \Gamma, \delta, \omega, q_s)$  where

$Q$  is a finite set of states  $\{q_i \mid i \text{ is a nonnegative integer}\}$

$\Sigma$  is the finite input alphabet

$\Gamma$  is the finite output alphabet

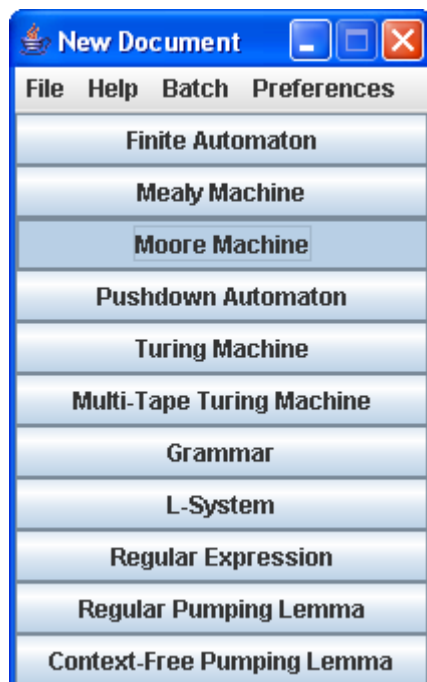
$\delta$  is the transition function,  $\delta : Q \times \Sigma \rightarrow Q$

$\omega$  denotes the output function,  $\omega : Q \rightarrow \Gamma$

$q_s$  (is a member of  $Q$ ) is the initial state

Moore machines are different than Mealy machines in the output function,  $\omega$ . In a Moore machine, output is produced by its states, while in a Mealy machine, output is produced by its transitions.

To start a new Moore machine, select the **Moore Machine** option from the main menu.



#### Starting a new Moore machine


## Differences between a Moore Machine and an FA

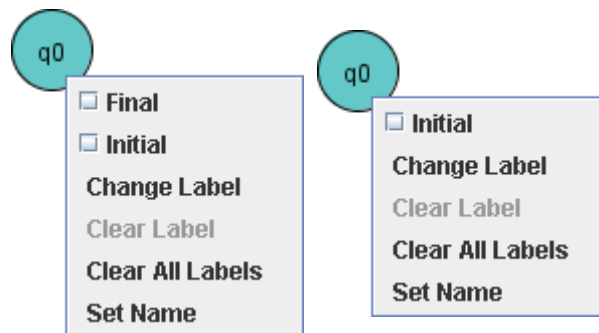
A Moore machine is very similar to a Finite Automaton (FA), with a few key differences:

- It has no final states.
- Its states produce output.
- It does not accept or reject input, instead, it generates output from input.
- Lastly, Moore machines cannot have nondeterministic states.

Let's go through these points.

### No Final State

In an FA, when you have the Attribute Editor tool selected (you may do so by clicking the  button), right-clicking on a state will produce a pop-up menu that allows you to, among other things, set it to be a final state. In a Moore machine, that option is not available.



An FA state menu

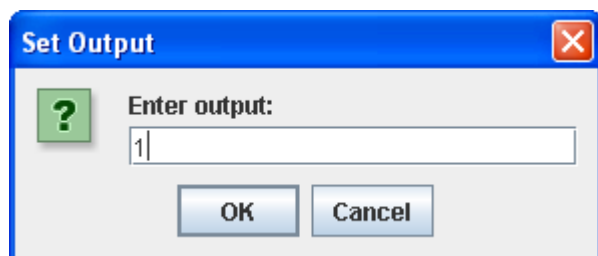
A Moore machine state menu

A Moore machine does not have final states because it does not accept or reject input. Instead, each state produces output, which will be described below.

### State Output

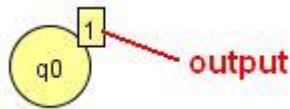
A Moore machine produces output when it is at a state.

Creating a Moore machine is the same as creating an FA with the exception of creating its states. In a Moore machine, each state produces output. When you are creating a state, a popup dialog box appears that prompts you for the output of the state.




Entering the state output

Type the state output in the dialog box and hit click **OK**. (If you click **Cancel**, the state will still be created, it will just have an empty string as its output.) When the state is created, its output will be show in its top right hand corner:



State created

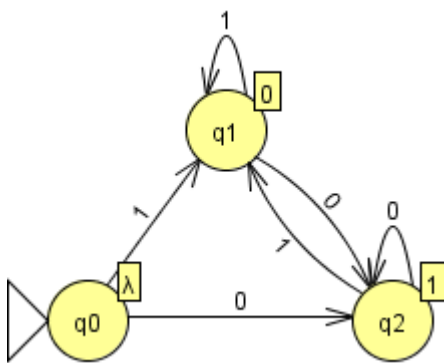
Thus, when the machine is in state  $q_0$ , it will produce an output of "1". To change the state output, click on the state using the Arrow tool , and enter the new output in the dialog box.

With each state producing output, the Moore machine can produce output from an input string.

### Producing Output from an Input String

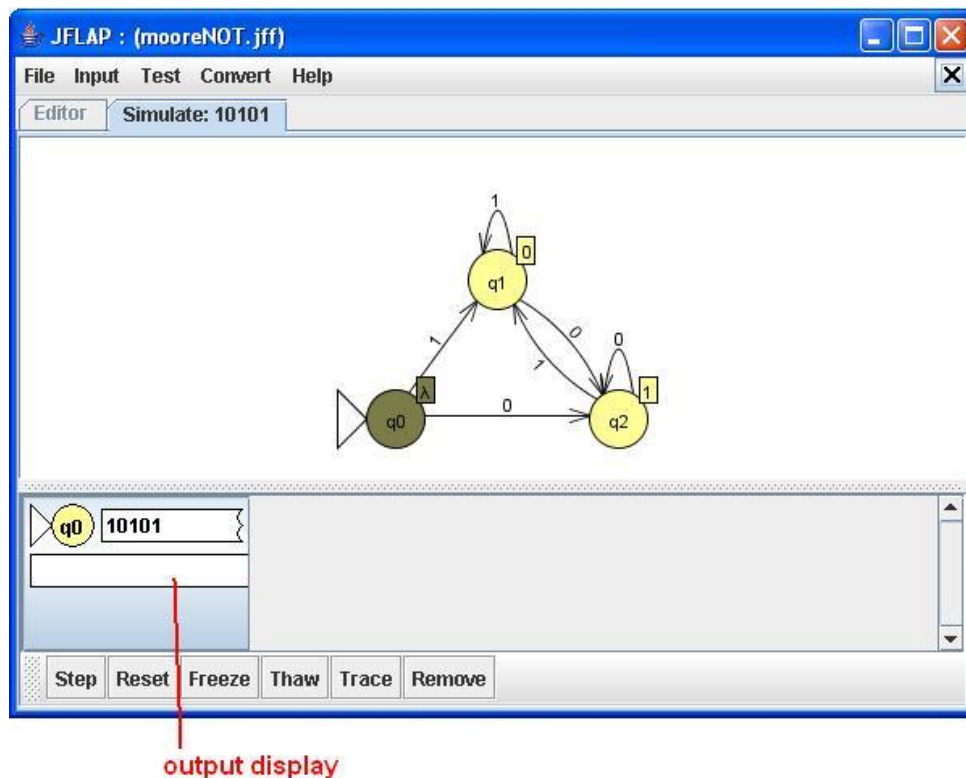
Instead of accepting or rejecting input, a Moore machine produces output from an input string.

Let's look at this simple Moore machine (which is provided in [mooreNOT.jff](#)):



A simple Moore machine

When we select the menu option **Input : Step...** and enter your input, we get a display similar to that of an FA, except with another piece of tape below the input tape. This displays the output of the machine at the current step.



The output display

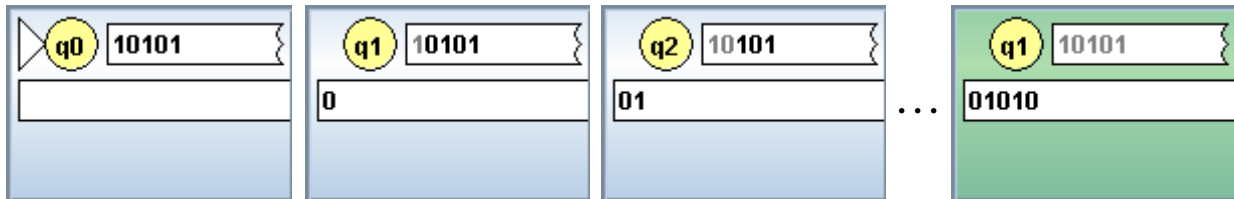
At each step. The output tape updates itself to display the total output that has been produced so far:

Initial:

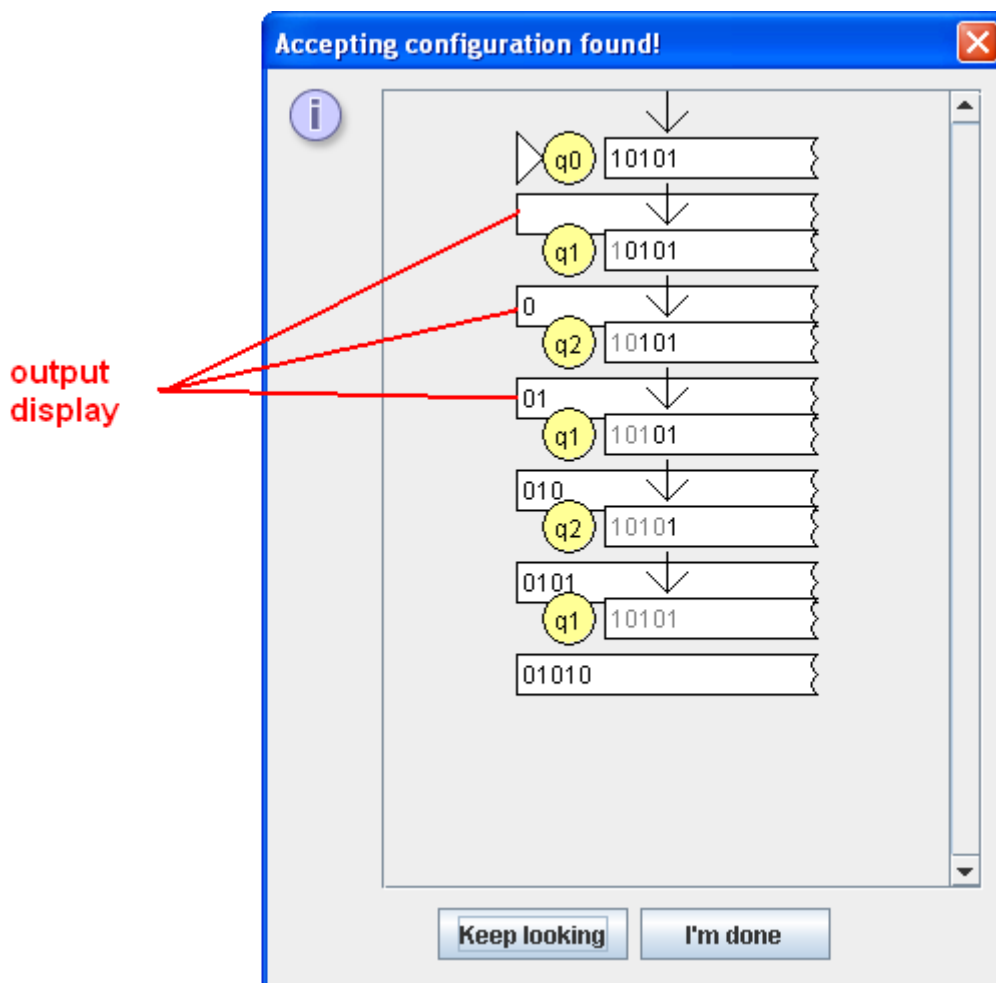
Step 1:

Step 2:

... Final:



The menu option **Input : Fast Run...** works similarly, with the output being displayed in a tape under the input tape:



Fast run output display

Similarly, when we select **Input : Multiple Run**, output is displayed in the **Result** column.



JFLAP : (mooreNOT.jff)

File Input Test Convert Help

Editor Multiple Run

The diagram shows a Moore machine with three states: q0 (start state), q1, and q2. Transitions are labeled with input/output pairs: q0 to q1 (1/0), q1 to q0 (0/1), q1 to q2 (0/1), q2 to q1 (1/0), q0 to q2 (0/1), and self-loops on q1 (1/0) and q2 (0/1).

Input	Result
10101	01010
11001	00110
101	010
00	11
1101	0010
1	0

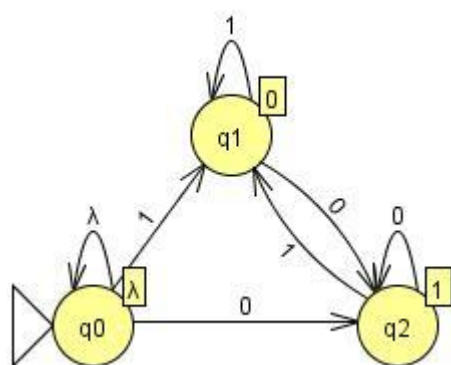
Run Inputs Clear Enter Lambda View Trace

Multiple run output display

### No Nondeterminism

As Moore machines map an input to a unique output, nondeterminism cannot exist in a Moore machine. Although we still will be able to build a Moore machine that has nondeterminism, we will not be able to run input on it.

For instance, let's modify our machine slightly to make:



Moore machine with nondeterminism

We we try to run it on an input, with **Input : Step...**, **Input : Fast Run...**, or **Input : Multiple Run...**, we will get an error message asking us to remove the nondeterministic states:



Select **Test : Highlight Nondeterminism** to view the nondeterministic states. Remove the nondeterminism in order to be able to run input on the Moore machine.

### Output Convention

In JFLAP, we adopt the output convention that the machine produces the output associated with the start state when it is turned on. (The alternate view is that the machine does not produce output until the first symbol of input is read.) If we wish to use the alternate view for a machine, we may just add a start state with the empty string as its output, like the machine described above.

## 10. Building Your First Pushdown Automaton

### Contents

#### Definition

#### How to Create an Automaton

#### Nondeterministic NPDAs

### Definition

JFLAP defines a nondeterministic pushdown automaton (NPDA)  $M$  as the septuple  $M = (Q, \Sigma, \Gamma, \delta, q_s, Z, F)$  where

$Q$  is a finite set of states  $\{q_i \mid i \text{ is a nonnegative integer}\}$

$\Sigma$  is the finite input alphabet

$\Gamma$  is the finite stack alphabet

$\delta$  is the transition function,  $\delta : Q \times \Sigma^* \times \Gamma^* \rightarrow \text{finite subsets of } Q \times \Gamma^*$

$q_s$  (is member of  $Q$ ) is the initial state

$Z$  is the start stack symbol (must be a capital Z)

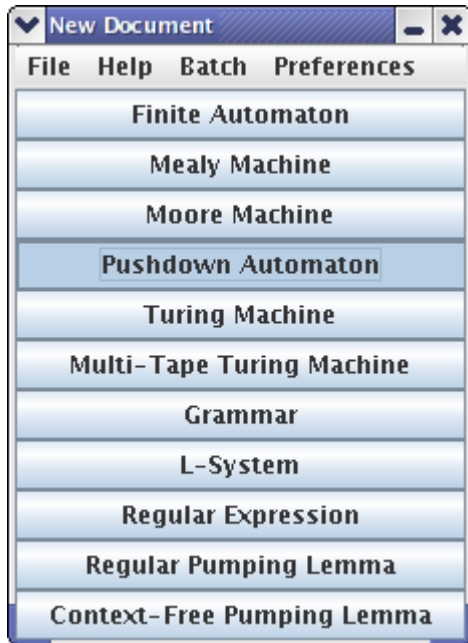
$F$  (is a subset of  $Q$ ) is the set of final states

Note that this definition includes deterministic pushdown automata, which are simply nondeterministic pushdown automata with only one available route to take.

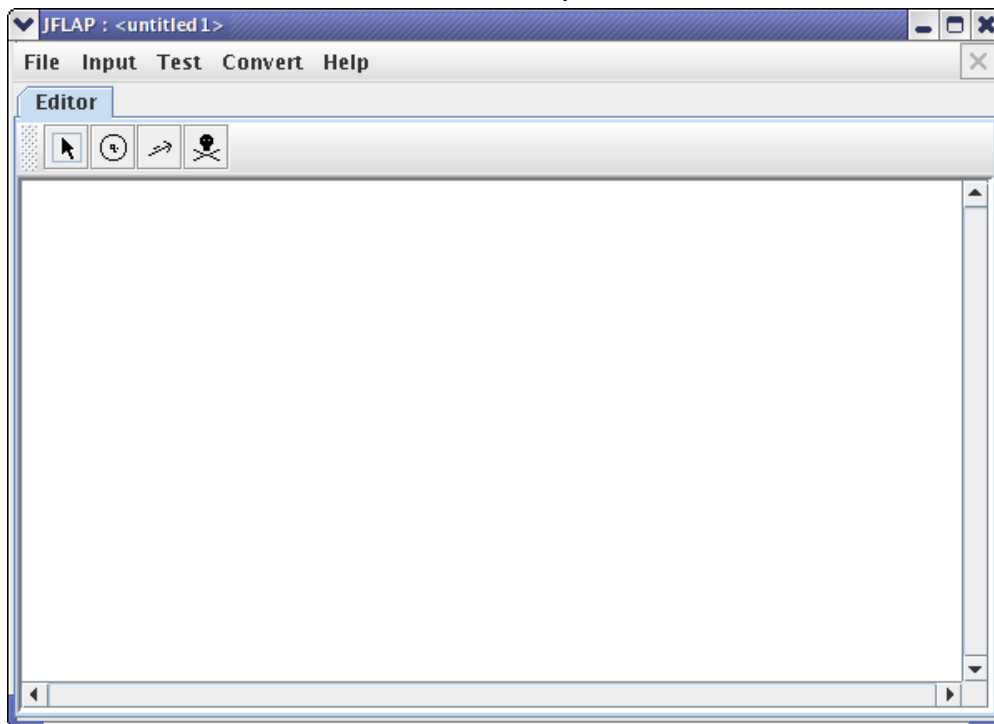
### How to Create an Automaton

For knowledge of many of the general tools, menus, and windows used to create an automaton, one should first read the tutorial on [finite automata](#). This tutorial will principally focus on features and options that differentiate pushdown automata from finite automata.

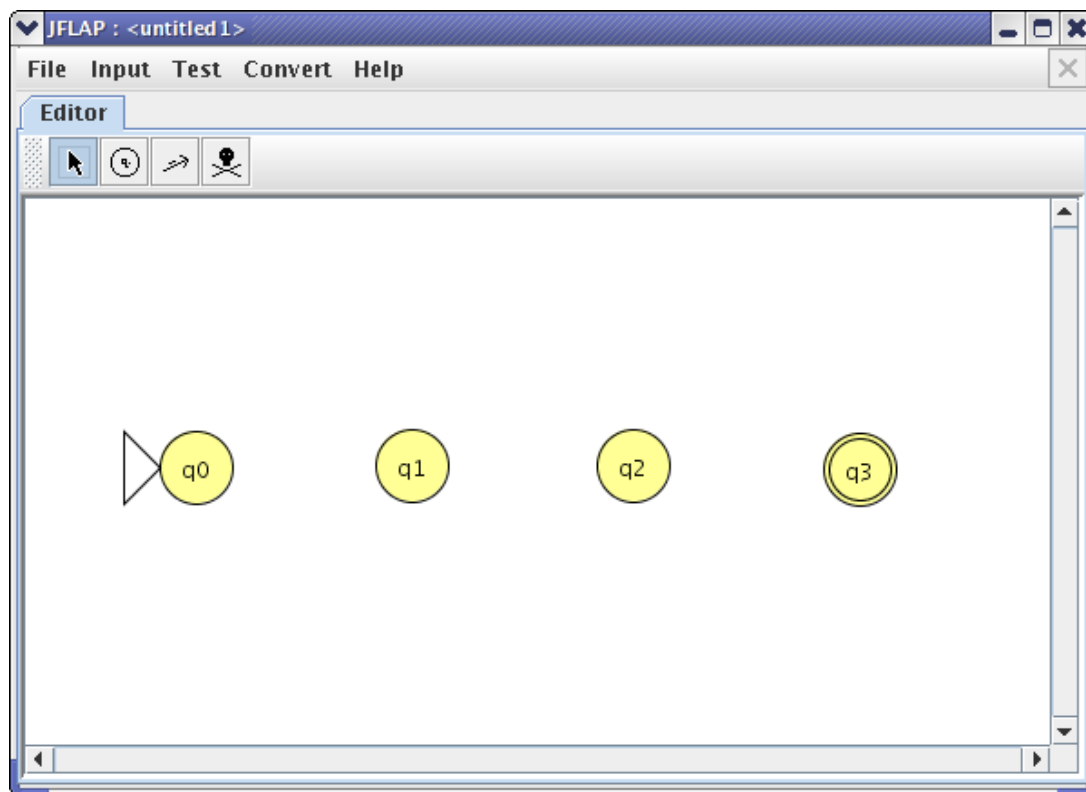
We will begin by constructing a deterministic NPDA for the language  $L = \{a^n b^n : n > 0\}$ . Our approach is to push an “a” on the stack for each “a” in the input, and to pop an “a” off the stack for each “b” in the input. To start a new NPDA, start JFLAP and click the **Pushdown Automaton** option from the menu, as shown below:



One should eventually see a blank screen that looks like the screen below. There are many of the same buttons, menus, and features present that exist for finite automata. However, there are a few differences, which we will encounter shortly.



Four states should be enough for the language  $L = \{a^n b^n : n > 0\}$ . Add four states to the screen, setting the initial state to be  $q_0$  and the final state to be  $q_3$ . The screen should look similar to one below.

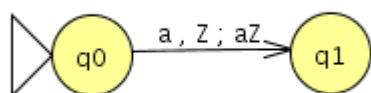


Now its time to add the transitions. Attempt to add a transition between the states  $q_0$  and  $q_1$ . However, there is something different about these transitions in comparison to those created with finite automata. One will notice that there are three inputs instead of one.

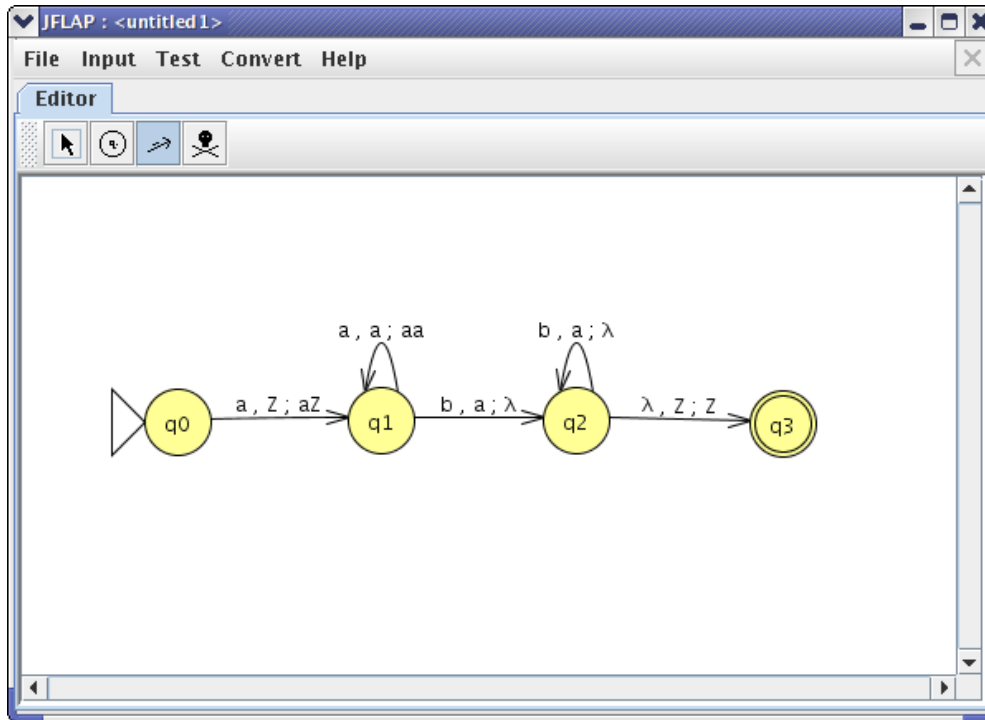


The value in the first box represents the input to be processed, the value in the second box represents the current value at the top of the stack, and the final value represents the new value to be pushed onto the top of the stack, after popping the value at the top of the stack off. There is no limit to the size of the values in any of these boxes.

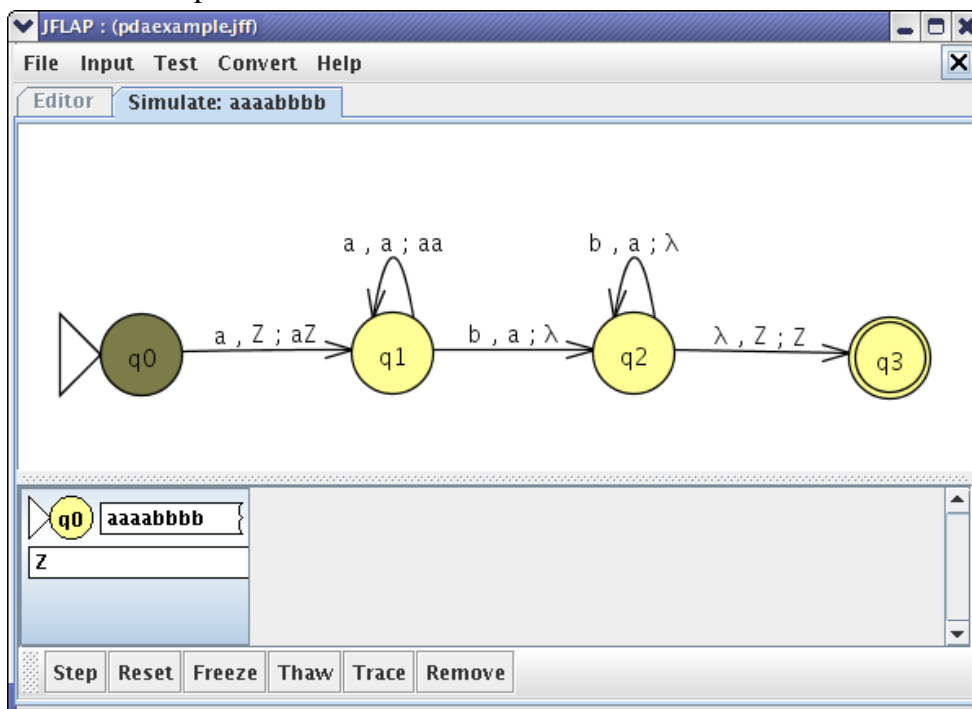
Now, it's time to add input. To change the transition from the default, click on the first box. Enter a value of “a” for the first box, a value of “Z” for the second box (the default character for the bottom element in the stack), and a value of “aZ” in the third box. Use Tab or the mouse to move between the boxes, and press enter or click the mouse on the screen outside the boxes when done. This transition means the the following. If the first symbol of the input is “a”, the first symbol of the stack is “Z”, and the machine is in state “ $q_0$ ”, then pop off “Z” and push “aZ” onto the stack. This transition thus adds an “a” to the stack if utilized. When done, the area between  $q_0$  and  $q_1$  should resemble the example below.



Let's finish up the transitions. Add a transition (a, a; aa) between  $q_1$  and  $q_1$  to finish up the  $a^n$  segment. Create the transition (b, a;  $\lambda$ ) between  $q_1$  and  $q_2$  and between  $q_2$  and  $q_2$  to represent the  $b^n$  segment. Finally, ( $\lambda$ , Z; Z) between  $q_2$  and  $q_3$  will allow an input to arrive at the final state. When finished, the screen should look like this:



For your convenience, the NPDA that we have just made is available in [pdaexample.jff](#). Now, click on the “Input” menu and select “Step with Closure.” It will prompt you for input, so enter “aaaabbbb” (representing  $a^4b^4$ ). After clicking “OK” or pressing enter, the following screen should come up:



This is very similar to the corresponding screen for finite automata. However, a noted difference is that there is a second box of text in the panel in the lower left. This text box, which currently contains a 'Z', is our stack. If we click the step button, we can see an example of how the stack changes as the simulation runs. There is now be an 'a' in front of the 'Z'. As we continue the simulation, the stack will be at its largest when the  $a^n$  values have been processed, but not any  $b^n$  values. However, as the  $b$  values are processed, it will get smaller. The stack will finally end back where it started, with a value of “Z”. Thus we finish our first deterministic NPDA.

After the first step



After  $a^n$



After the first  $b$  value

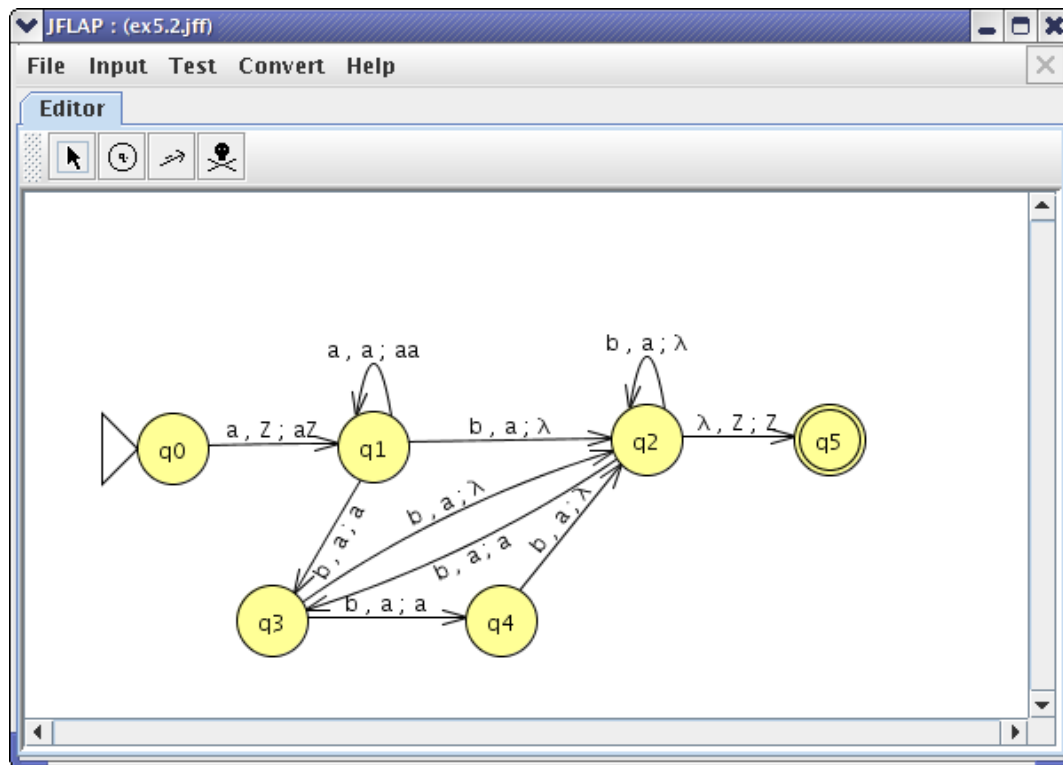


When finished



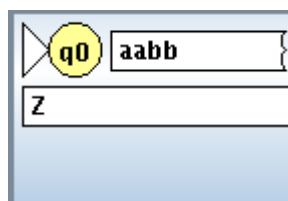
### Nondeterministic NPDAs

Nondeterministic NPDAs work in a similar way. One example is the one below, which one can access online here [here](#).

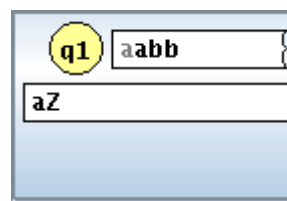


The following is a simulation of “Input  $\rightarrow$  Step by Closure” with the string “aabb”:

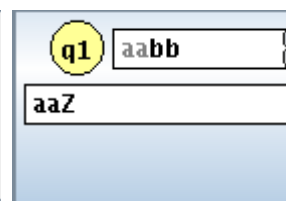
Start



Step 1



Step 2



Now is when the nondeterminism begins to assert itself, and the automaton can take more than one path. The last steps cover all possible permutations the program can take, with one path succeeding in the end.

### Step 3

q3 aabb	q2 aabb
aaZ	aZ

### Step 4

q3 aabb	q2 aabb	q2 aabb	q4 aabb
aZ	Z	aZ	aaZ

### Results

q4 aabb	q2 aabb	q3 aabb	q5 aabb
aaZ	aZ	aZ	Z

(Note: This example taken from *JFLAP: An Interactive Formal Languages and Automata Package* by Susan Rodger and Thomas Finley.)



## 11. Converting a NPDA to a Context-Free Grammar

### Contents

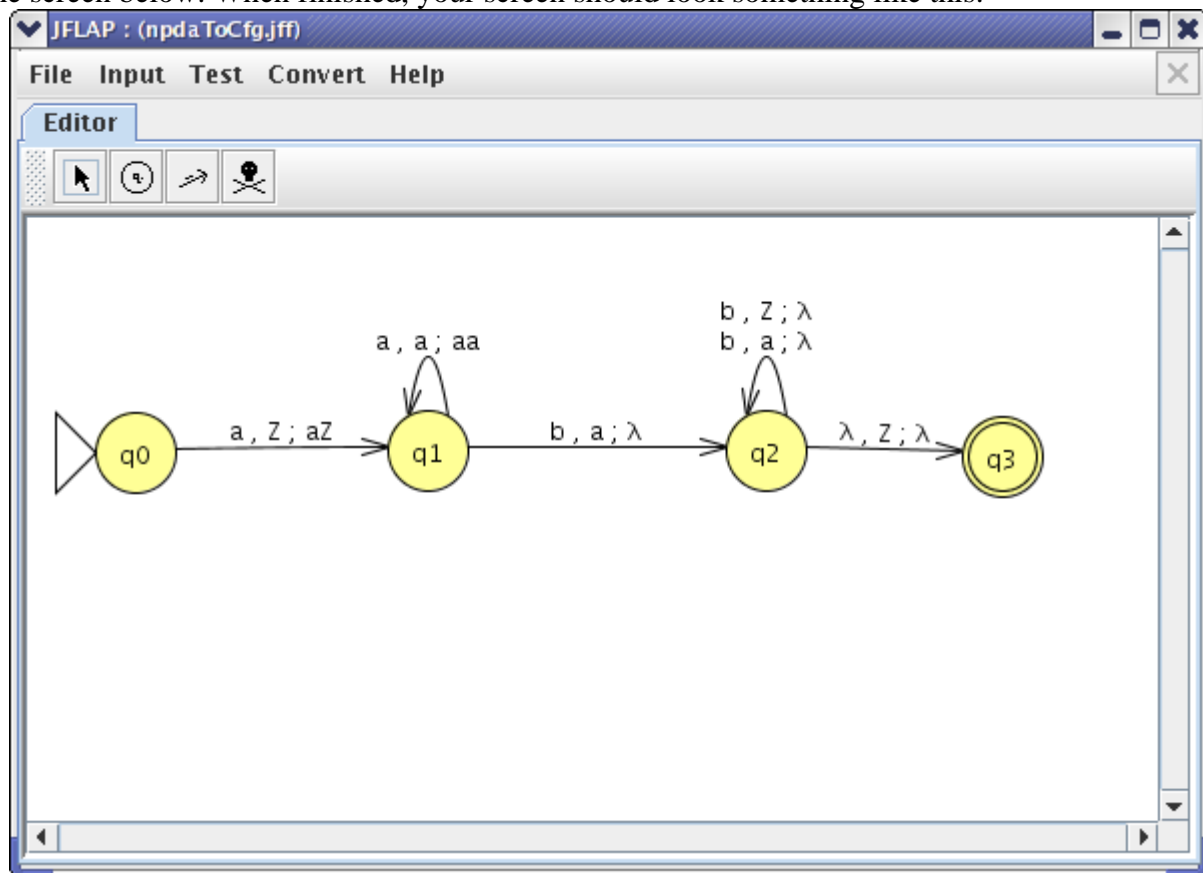
[Introduction](#)

[Converting to a Context-Free Grammar](#)

[An Exportable Example](#)

### Introduction

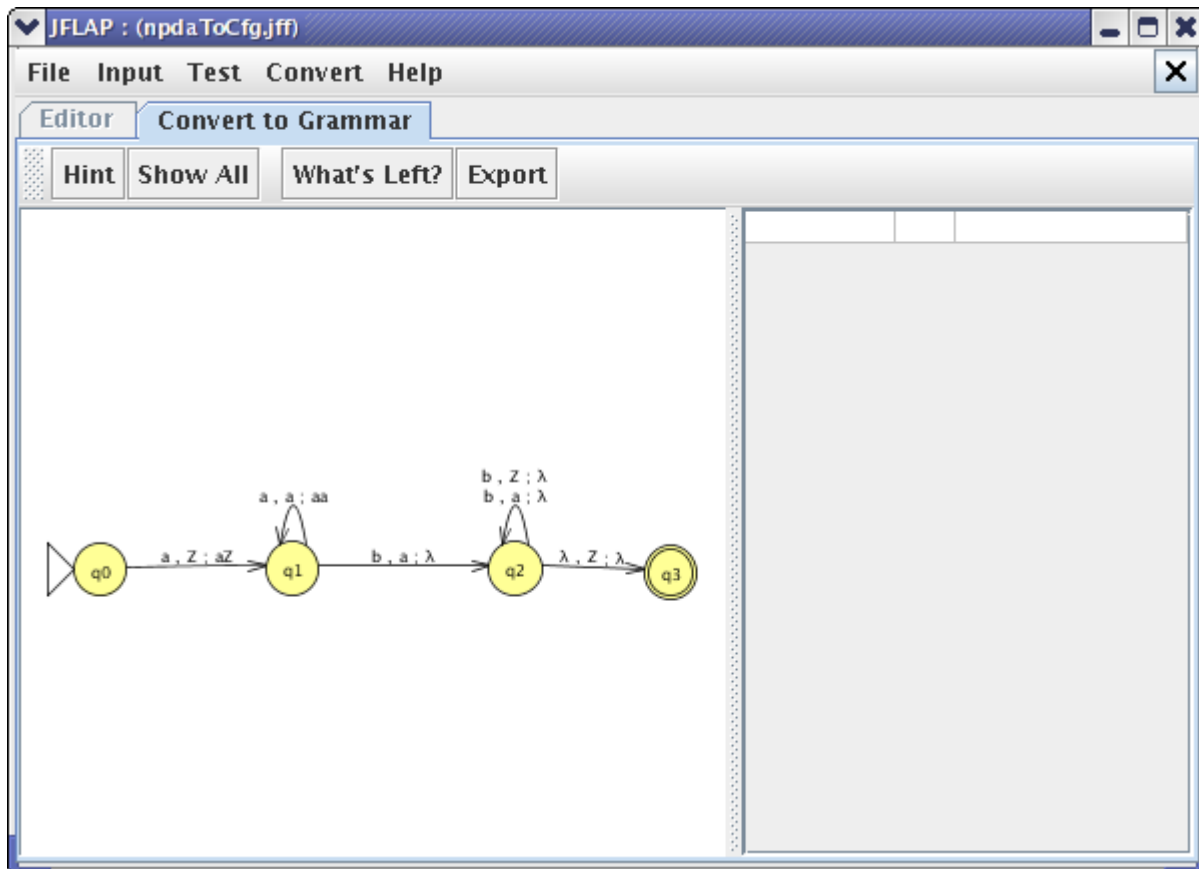
It is recommended, if you haven't already, to read the tutorial about [creating a pushdown automaton](#), which covers the basics of constructing a NPDA. This section specifically describes how one may transform any nondeterministic pushdown automaton (NPDA) into a context-free grammar by using the tools under the “Convert → Convert to Grammar” menu option. To get started, open JFLAP. Then, either load the file [npdaToCfg.jff](#), or construct the NPDA present in the screen below. When finished, your screen should look something like this:



The idea in the transformation is to convert each transition into one or more productions that mimic the transitions. In some cases, many possible productions are generated in order to generate the equivalent productions, resulting in many useless productions.

### Converting to a Grammar

We will now convert this NPDA into a CFG. Click on the “Convert → Convert to Grammar” menu option, and this screen should come up:



It should be noted, before continuing, that in order for JFLAP's conversion algorithm to work, all transitions must pop 1 symbol off the stack and push onto it either 2 or 0 symbols. In addition, there can be only one final state, and any transition into the final state must pop Z off the stack. If you try to proceed to the “Convert → Convert to Grammar” menu option with an invalid state or condition, then an error message will come up. Once all illegal transitions and states have been fixed, JFLAP will proceed to the screen above. Fortunately, our example is a legal candidate for conversion, so we can proceed.

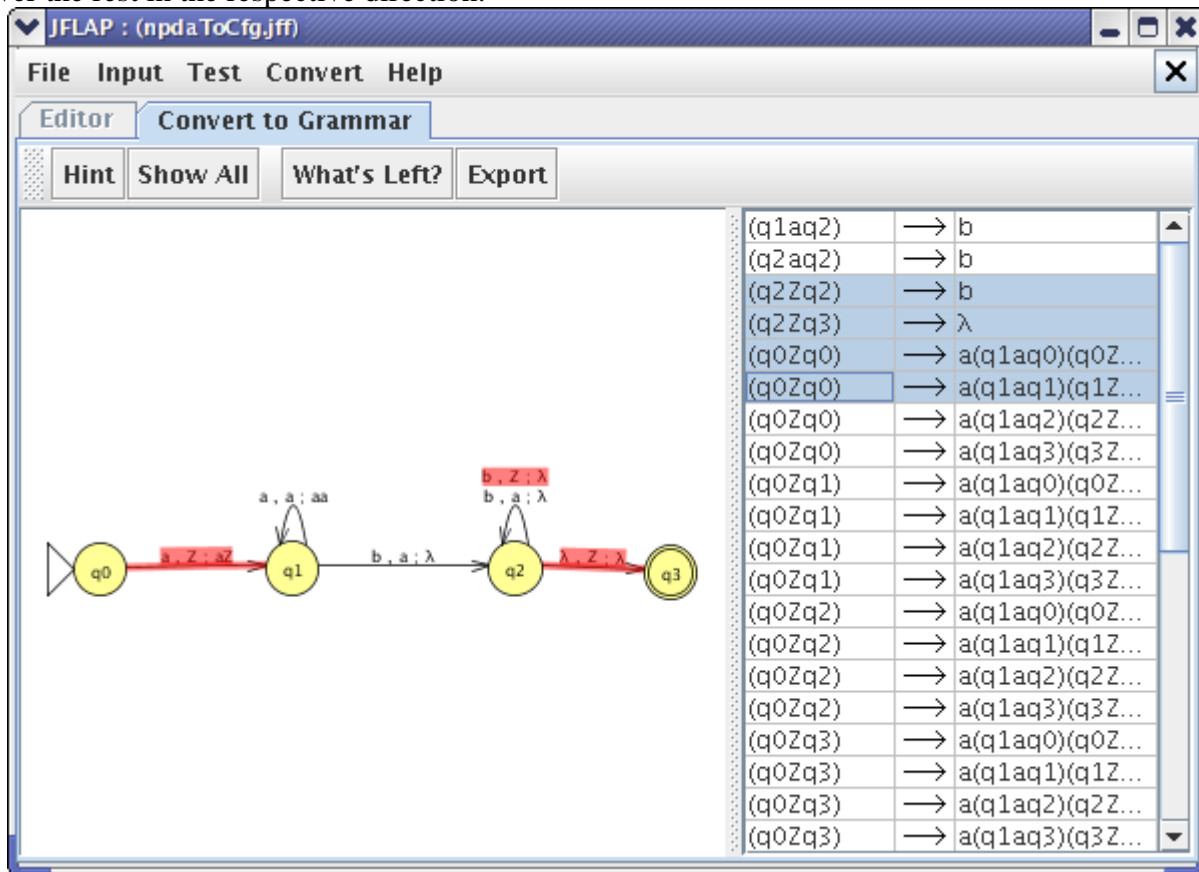
All we need to do to proceed is to click on the specific transitions on our automaton, and all possible grammar rules that can be generated from it will emerge on the right. First, let's add the transitions that pop off an item and then push nothing onto the stack, since they only generate one rule. Click on the last four transitions, and eventually, in the list of rules to the right, you should see some rules resembling this:

(q1aq2)	→	b
(q2aq2)	→	b
(q2Zq2)	→	b
(q2Zq3)	→	λ

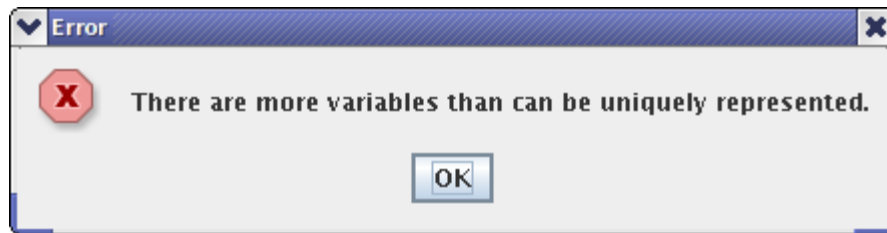
Now let's add the last two transitions. First, click on the transition leading from “q0” to “q1”. Instead of just one neat grammar rule, however, we are presented with a large list of grammar rules.

(q0Zq0)	→	a(q1aq0)(q0Z...
(q0Zq0)	→	a(q1aq1)(q1Z...
(q0Zq0)	→	a(q1aq2)(q2Z...
(q0Zq0)	→	a(q1aq3)(q3Z...
(q0Zq1)	→	a(q1aq0)(q0Z...
(q0Zq1)	→	a(q1aq1)(q1Z...
(q0Zq1)	→	a(q1aq2)(q2Z...
(q0Zq1)	→	a(q1aq3)(q3Z...
(q0Zq2)	→	a(q1aq0)(q0Z...
(q0Zq2)	→	a(q1aq1)(q1Z...
(q0Zq2)	→	a(q1aq2)(q2Z...
(q0Zq2)	→	a(q1aq3)(q3Z...
(q0Zq3)	→	a(q1aq0)(q0Z...
(q0Zq3)	→	a(q1aq1)(q1Z...
(q0Zq3)	→	a(q1aq2)(q2Z...
(q0Zq3)	→	a(q1aq3)(q3Z...

Now, it's possible not every one of these rules will be useful. The correct ones are present, however. The multiplicity of rules is a result of generating with brute force all possible stack combinations. Finally, click on the final transition to add the rules associated with it. When done, your screen may resemble the one below. The highlighted transitions correspond to the selected rules on the right. In order to select a set rule, click on the first or last rule and drag the mouse over the rest in the respective direction.

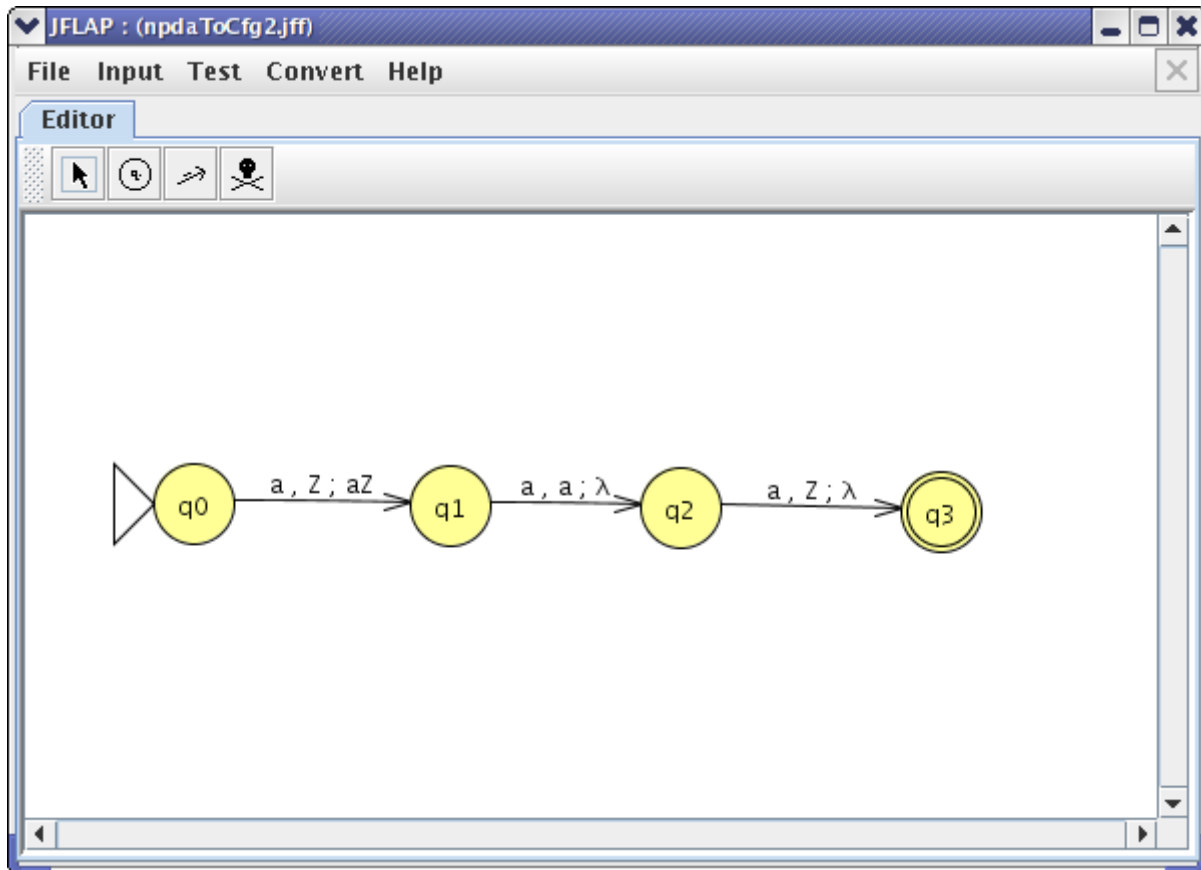


You may wish to know what the buttons in the toolbar do, since we have not yet used them. The "Hint" button will add the rules of one transition to the list that haven't yet been added. The "Show All" button will add the rules of all remaining transitions to the list. The "What's Left" button will highlight all the transitions that haven't yet been selected. The "Export" button will generate a new context-free grammar using a newly complete rules list in the right panel. However, in the present version of JFLAP, the button will not work for this example because the example generates too many variables. The following is the window that will come up if you try to export the file.



### An Exportable Example

Let's now try a file that is exportable. Either load the file [npdaToCfg2.jff](#), or construct the NPDA present in the screen below in a new window. When finished, your screen should look something like this:



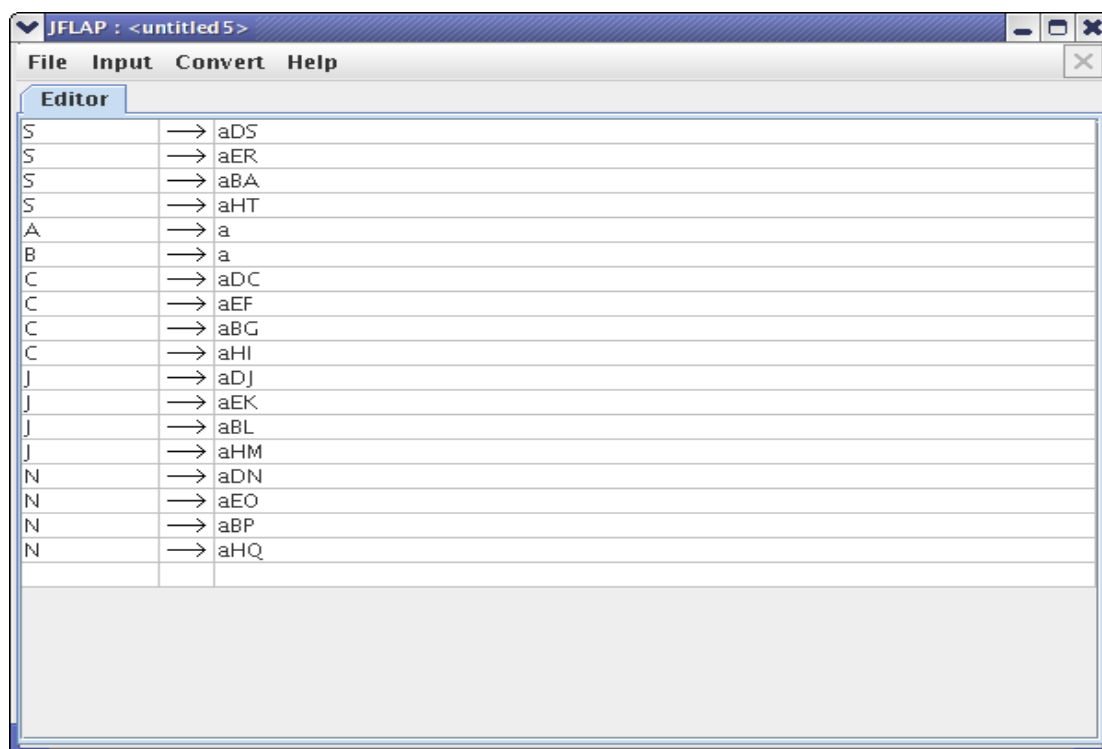
You can see, both by examining the NPDA and noticing some of the strings that it accepted or rejected in the list to the left, that the NPDA accepts the language “aaa”. Using the same tools to generate the grammar as before, when finished, you should end up with a list of rules resembling the list on the right:

## Sample Automaton Input (from Input → Complete List of Rules Multiple Run)

	Reject
a	Reject
aa	Reject
aaa	Accept
aaaa	Reject

(q2Zq3)	→	a
(q1aq2)	→	a
(q0Zq0)	→	a(q1aq0)(q0Zq0)
(q0Zq0)	→	a(q1aq1)(q1Zq0)
(q0Zq0)	→	a(q1aq2)(q2Zq0)
(q0Zq0)	→	a(q1aq3)(q3Zq0)
(q0Zq1)	→	a(q1aq0)(q0Zq1)
(q0Zq1)	→	a(q1aq1)(q1Zq1)
(q0Zq1)	→	a(q1aq2)(q2Zq1)
(q0Zq1)	→	a(q1aq3)(q3Zq1)
(q0Zq2)	→	a(q1aq0)(q0Zq2)
(q0Zq2)	→	a(q1aq1)(q1Zq2)
(q0Zq2)	→	a(q1aq2)(q2Zq2)
(q0Zq2)	→	a(q1aq3)(q3Zq2)
(q0Zq3)	→	a(q1aq0)(q0Zq3)
(q0Zq3)	→	a(q1aq1)(q1Zq3)
(q0Zq3)	→	a(q1aq2)(q2Zq3)
(q0Zq3)	→	a(q1aq3)(q3Zq3)

Export the file, and you should have roughly the following list of productions. Those productions that generate useless variables that do not lead to anything are never used, although they are generated by the JFLAP algorithm:



The screenshot shows the JFLAP application window with the title "JFLAP : <untitled5>". The menu bar includes "File", "Input", "Convert", and "Help". The "Editor" tab is active, displaying a list of productions. Each production is on a new line, consisting of a variable on the left, followed by an arrow "→", and a string of variables and terminals on the right. The productions are:

S	→	aDS
S	→	aER
S	→	aBA
S	→	aHT
A	→	a
B	→	a
C	→	aDC
C	→	aEF
C	→	aBG
C	→	aHI
J	→	aDJ
J	→	aEK
J	→	aBL
J	→	aHM
N	→	aDN
N	→	aEO
N	→	aBP
N	→	aHQ

Feel free to use the brute force parser on this grammar with the input string “aaa” to see that the string is accepted.

## 12. Building A Turing Machine

### Contents

[Definition](#)

[How to Create a Turing Machine](#)

[Using Your New Machine as a Building Block](#)

[Transitions from Final States](#)

[Shortcut Syntax for Turing Machines](#)

### Definition

JFLAP defines a Turing Machine  $M$  as the septuple  $M = (Q, \Sigma, \Gamma, \delta, q_s, \square, F)$  where

$Q$  is the set of internal states  $\{q_i \mid i \text{ is a nonnegative integer}\}$

$\Sigma$  is the input alphabet

$\Gamma$  is the finite set of symbols in the tape alphabet

$\delta$  is the transition function

$S$  is  $Q * \Gamma^n \rightarrow \text{subset of } Q * \Gamma^n * \{L, S, R\}^n$

$\square$  is the blank symbol.

$q_s$  (is member of  $Q$ ) is the initial state

$F$  (is a subset of  $Q$ ) is the set of final states

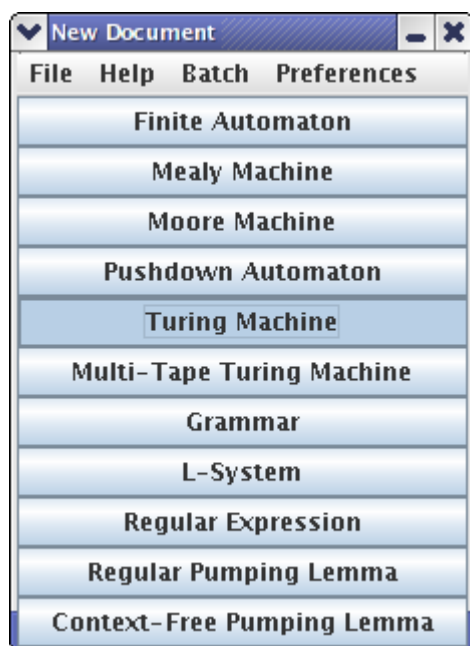
Note that this definition includes both deterministic and nondeterministic Turing machines.

### How to Create a Turing Machine

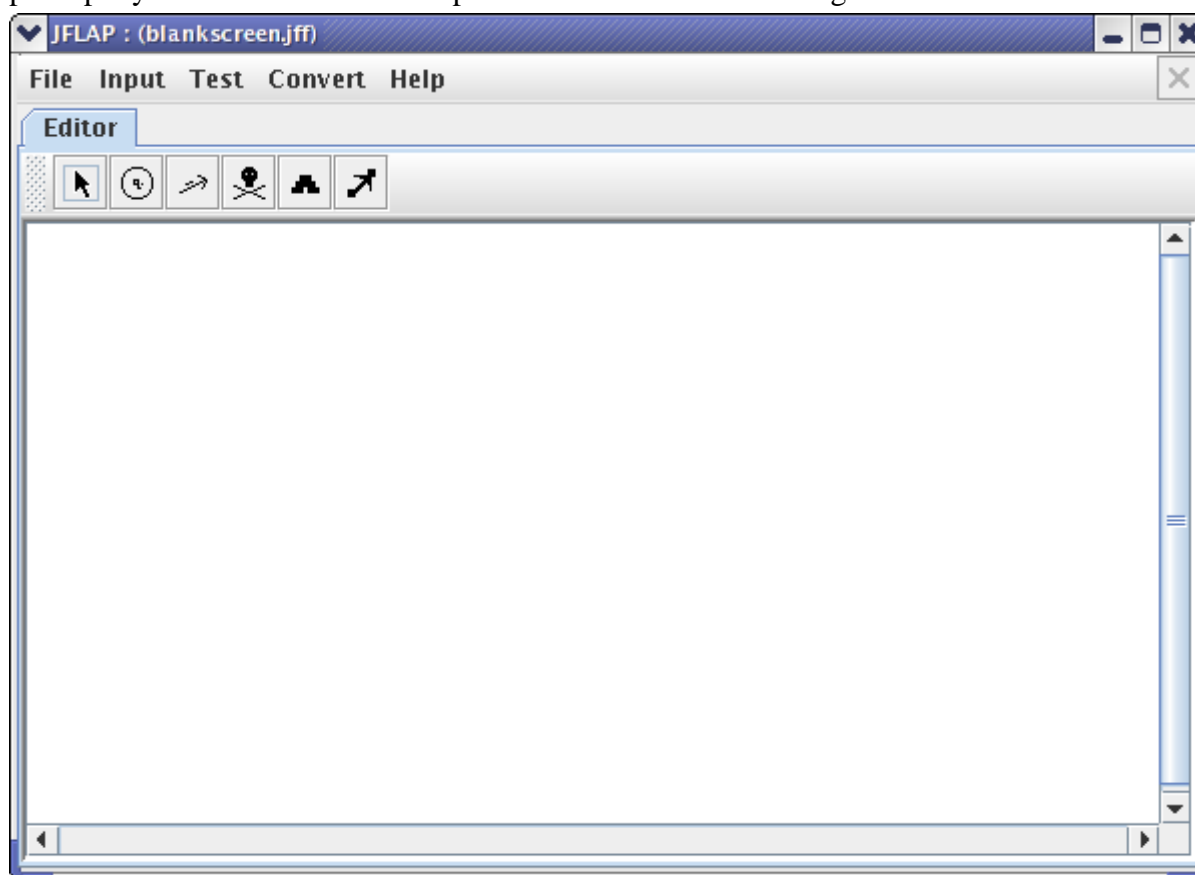
For knowledge of many of the general tools, menus, and windows used to create an automaton, one should first read the tutorial on [finite automata](#). This tutorial will principally focus on features and options differing from the finite automaton walkthrough, and offer an example of constructing a Turing machine.

Before starting, click on the “Preferences” item in the menu. A few preferences will be listed, and one of them, with a check box next to it, is “Enable Transitions from Turing Machine Final States.” Don't do anything with this preference just now and leave it unchecked, but just note that it exists.

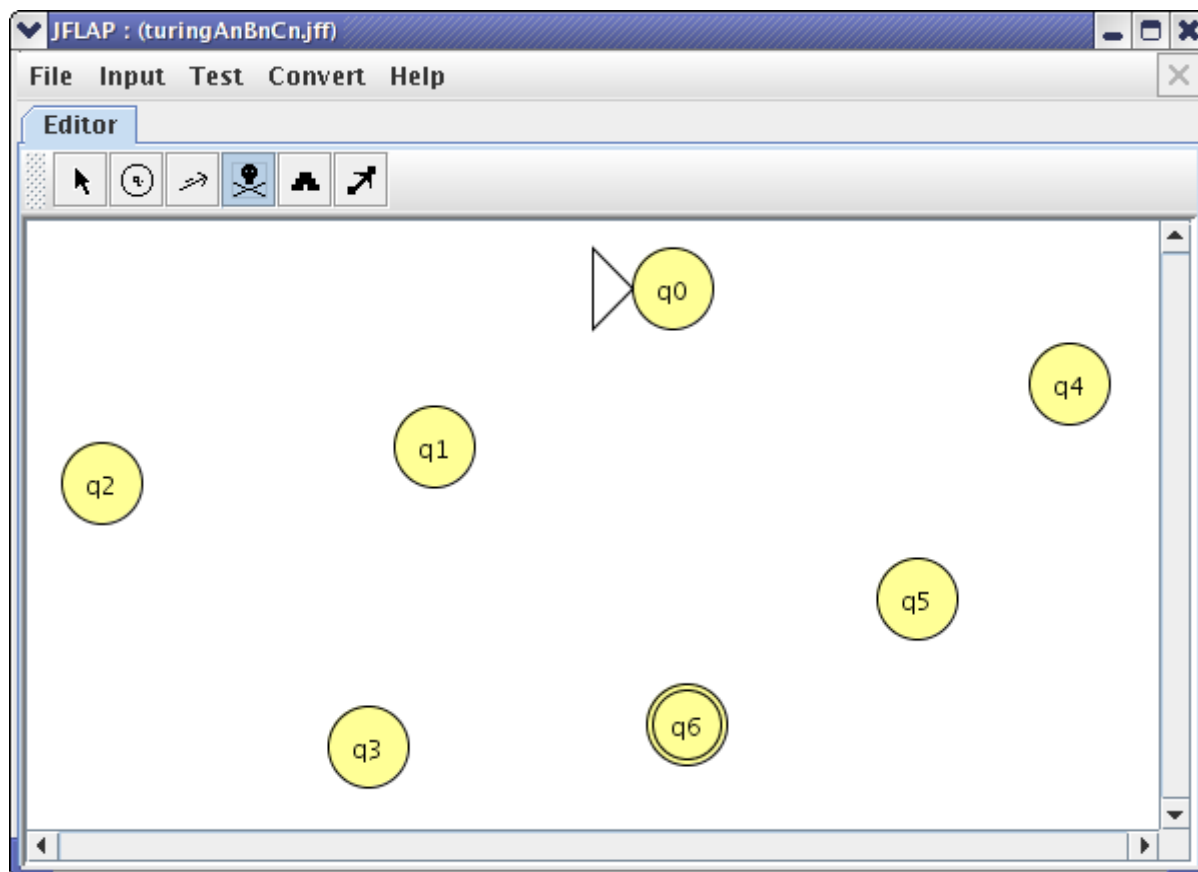
We will begin by constructing a Turing machine for the language  $L = \{a^n b^n c^n\}$ . To start a new one-tape Turing machine, start JFLAP and click the **Turing Machine** option from the menu, as shown below:



One should eventually see a blank screen that looks like the screen below. There are many of the same buttons, menus, and features present that exist for finite automata. This tutorial will principally focus on features and options that differentiate Turing machines from finite automata.



We will be adding a lot of states to create a Turing machine for  $L = \{a^n b^n c^n\}$ . Add seven states to the screen, setting the initial state to be  $q_0$  and the final state to be  $q_6$ . The screen should be roughly similar to one below.

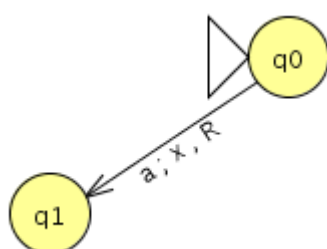


Now its time to add the transitions. Attempt to add a transition between the states q0 and q1. However, there is something different about these transitions in comparison to those created with finite automata. One will notice that there are three inputs instead of one.

<input type="text"/>	<input type="text"/>	R
----------------------	----------------------	---

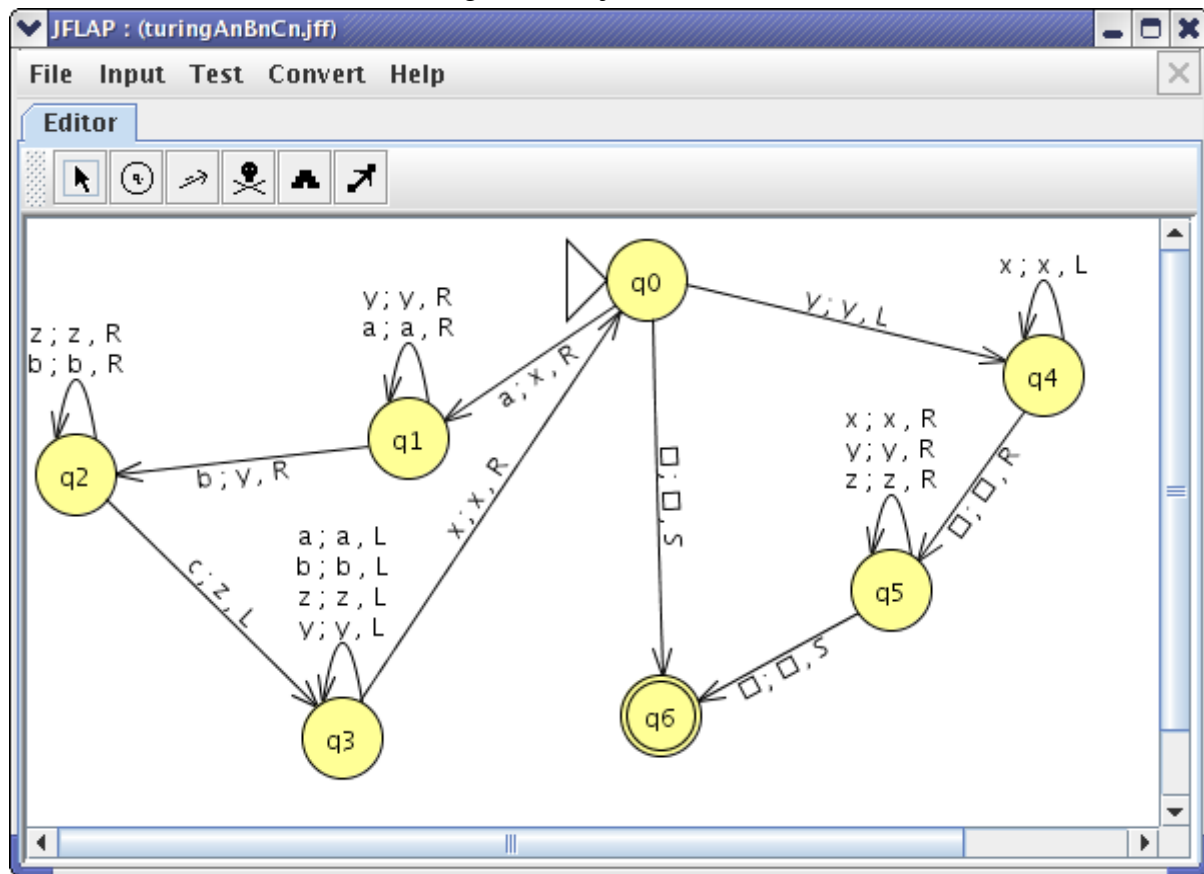
The value in the first box represents the current value under the head of the Turing machine. The second value is the value that will replace the first value on the tape once this step has been processed. The size of the values in these two boxes is limited to one character. The third value represents where the head will move after processing the step. It can be one of three values: 'R' (move right one square), 'L' (move left one square), and S (stay put and do not move the head). One could enter the value directly, or enter it from the pull-down menu that comes up when the third box is clicked on directly.

Now, it's time to add input. To change the transition from the default, click on the first box. Enter a value of "a" for the first box, a value of "x" for the second box, and a value of "R" in the third box. Use Tab or the mouse to move between the boxes, and press enter or click the mouse on the screen outside the boxes when done. This transition has the following meaning. If the head is under an "a" and the machine is in state "q0", then replace the "a" with an "x" and move the head to the right. When done adding input, the area between q0 and q1 should resemble the example below.

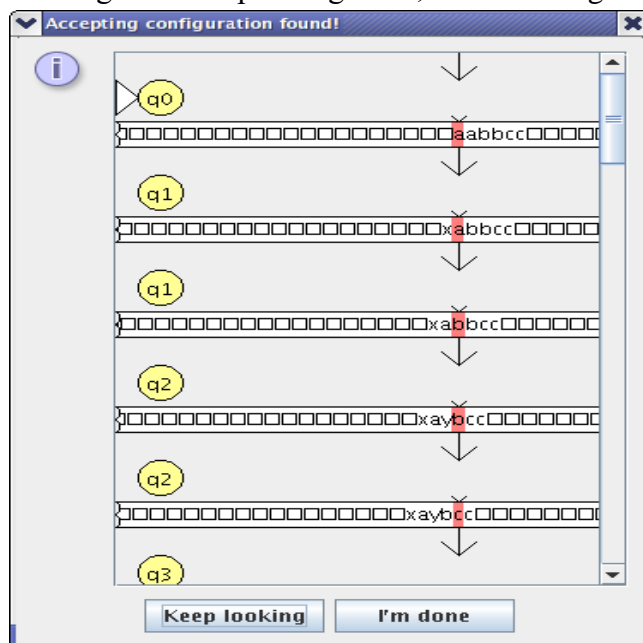




Let's finish up the transitions. Add the transitions in your screen below to your Turing machine. If you would rather not add every transition directly and would prefer to load the file of the screen below, it is available at [turingAnBnCn.jff](#).



Now, let's try out our new Turing machine. Because of the number of steps, we will avoid the “Step” option we used with finite automata (although for finite automata titled “Step with Closure”) and instead use the “Fast Run” option. To use this, click on the “Input” menu, and then click on “Fast Run”. When it prompts you for input, enter “aabbcc”, representing  $a^2b^2c^2$ . After clicking “OK” or pressing enter, the following screen should come up:

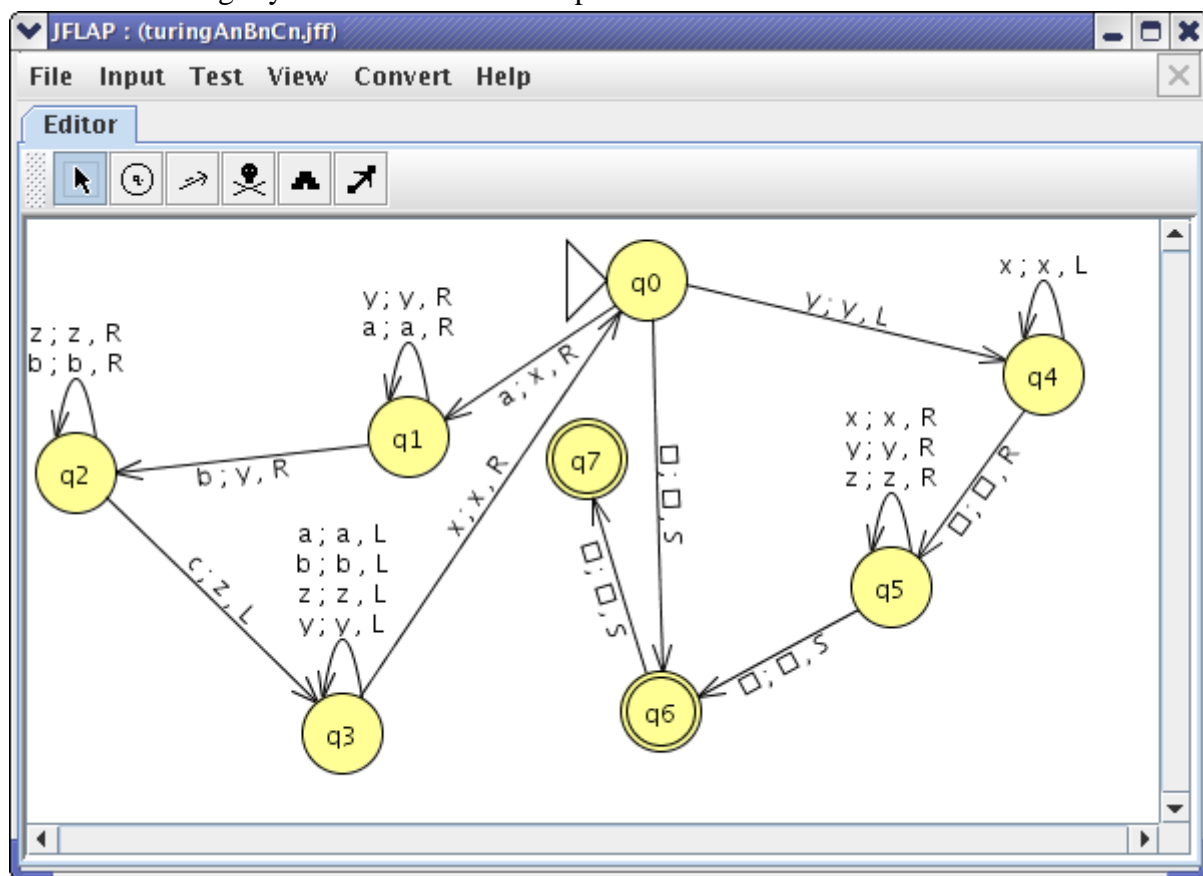


One can scroll down and see the tape, the current state, and the position of the head as the automaton processes the input step by step. One can see the algorithm at work, which is if the head encounters an “a”, it replaces it with an “x”. Then, it replaces a corresponding “b” with a “y” and a corresponding “c” with a “z”. This repeats until it is no longer possible, and this loop is what makes up the cycle encompassing “q0”, “q1”, “q2”, and “q3”. Once this is done, the program makes sure that there is nothing but “x”s, “y”s, and “z”s left in the correct order. In the case of input with length zero, the program immediately goes to the final state.

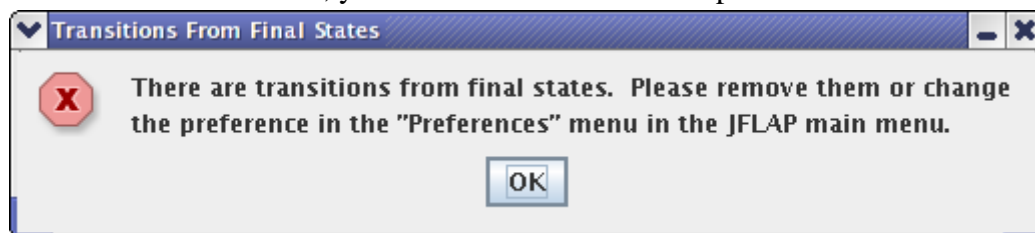
The “Keep looking” button is for finding other possible paths in automata that aren't deterministic, which is not applicable here. When finished, click “I'm done.” Congratulations, you have built your first Turing Machine!

### Transitions from Final States

Recall the “Enable Transitions from Turing Machine Final States” preference mentioned earlier. Also note the slightly modified earlier example below that now has two final states.



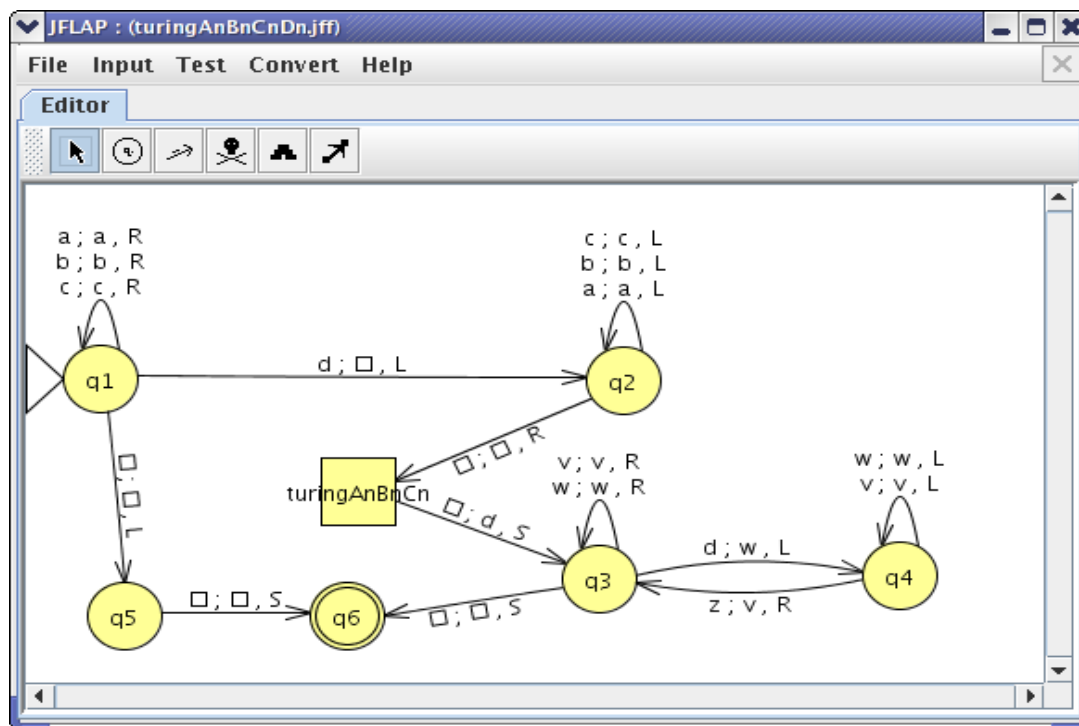
Because the preference was not enabled, and because there is an edge leading from the final state “q6”, the following error message will appear if you try to run it. Just note that if you wish to simulate such a machine, you need to either enable the preference or remove all offending edges.



## Using Your New Turing Machine as a Building Block

There are a few other features in JFLAP concerning Turing machines, and one very useful one is “building blocks”. We will not go into an in depth study of building blocks on this page, and one can learn more about them [here](#). However, it is worth noting that Turing machines, once created, can function as building blocks in other machines. Below is one such example, where the block we just created for  $L = \{a^n b^n c^n\}$  is used to implement the language  $L = \{a^n b^n c^n d^n\}$ . One can create Turing machines to accomplish one task, and if another task could utilize the first task to further its designs, one can use a building block as a shortcut to represent that task on the screen. The “block” was put onto the screen by the second rightmost button in the toolbar, which has an icon resembling a step pyramid. When clicked, a file menu will come up, as if you were opening a file. By selecting the file for the language  $L = \{a^n b^n c^n\}$ , and clicking “Open”, a yellow square will appear on the screen labeled by the file name. This functions on the screen as a state, and transitions can be created to and from it. Whenever something leads to the building block, it will preform its task based on the current state of the tape, and the tape will be changed by the block's output, for the benefit of the rest of the states in the machine.

The screen below is an example of using a block for  $L = \{a^n b^n c^n\}$  as a tool to implement the language  $L = \{a^n b^n c^n d^n\}$ , this example is accessible [here](#):



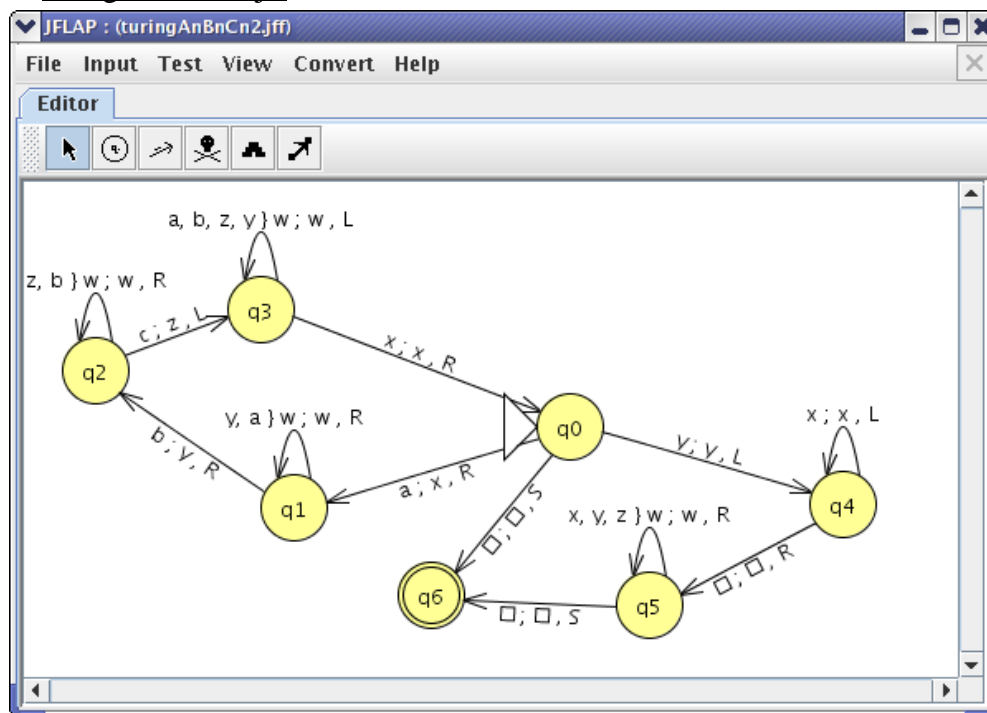
One may try a variety of different inputs and realize that the block functions as an acceptor by checking to see whether the number of “a”s, “b”s, and “c”s equal at the beginning of the input. It also functions as a transducer, as it changes those values to “x”s, “y”s, and “z”s respectively. The program builds off of this by checking to see whether the number of “d”s equals the number of “z”s. One must also note that a blank was placed where the first “d” value was. This is because, in order for the  $a^n b^n c^n$  acceptor to work, one must have blanks surrounding the smaller string. The “d” value is restored after leaving the block.

## Shortcut Syntax for Turing Machines

It is worth mentioning a few shortcut syntax rules that JFLAP implements. These syntax rules were developed for use with building blocks, but they also can be used with standard Turing Machines. These rules are covered in depth in the building blocks tutorial, but for the sake of a general overview, a brief summary is as follows. The first shortcut is that there exists the option of using the “!” character to convey the meaning of “any character but this character.” For example, concerning the transition (!a; x, R), if the head encounters any character but an “a”, it will replace the character with an “x” and move right. To write the expression “!□”, just type a “!” in when inputting a command.

One can also utilize variables when constructing a Turing machine in order to make inputting rules less tedious. For example, there is a second special character, “~”, that stands for whichever character was last read. Thus, in the transition (~; ~, R), the head will, no matter what character is underneath it, move to the right without changing the character. One can also explicitly define other variables. For example, the transition (a,b,c)w; w, R) would command the head, if either “a”, “b”, or “c” was under it, to assign the letter to the new variable *w* and then to move right without changing the tape. Whenever *w* is encountered later in the machine, it is synonymous with its stored value until it is assigned another value.

Our Turing machine from earlier is shown below in a slightly different layout with some variable-containing transitions (there was not a good place to use the “!” feature). Notice the fewer transitions present for performing the exact same task. This machine is available in [turingAnBnCn2.jff](#).



## 13. Multi-Tape Turing Machines

### Contents

[Introduction](#)

[How to Create an Multi-Tape Turing Machine](#)

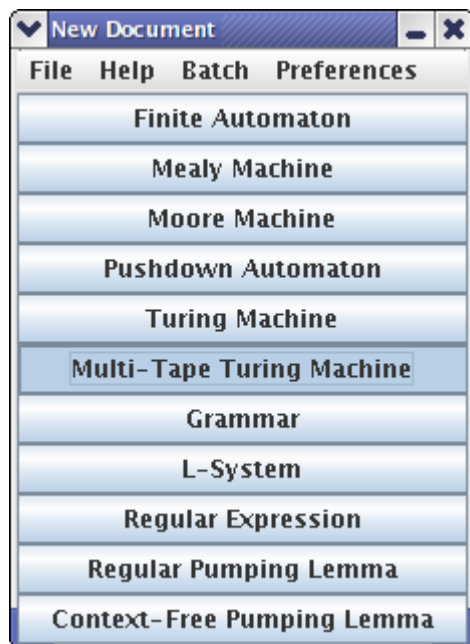
### Introduction

It is recommended, if you haven't already, to read the tutorial about [creating a one-tape Turing machine](#). It covers much of the basics about Turing machines and how their displays are different from other automata.

Multi-tape Turing machines as implemented in JFLAP are Turing machines ranging from 2 to 5 tapes. Values on both tapes must match for the automaton to proceed, but other than that, there isn't much difference between the two modes. Multi-tape Turing machines allow for a number of languages to be implemented with greater ease. One such example is the language demonstrated in the one tape tutorial,  $L = \{a^n b^n c^n\}$ . We will build this example with 3 tapes and test it.

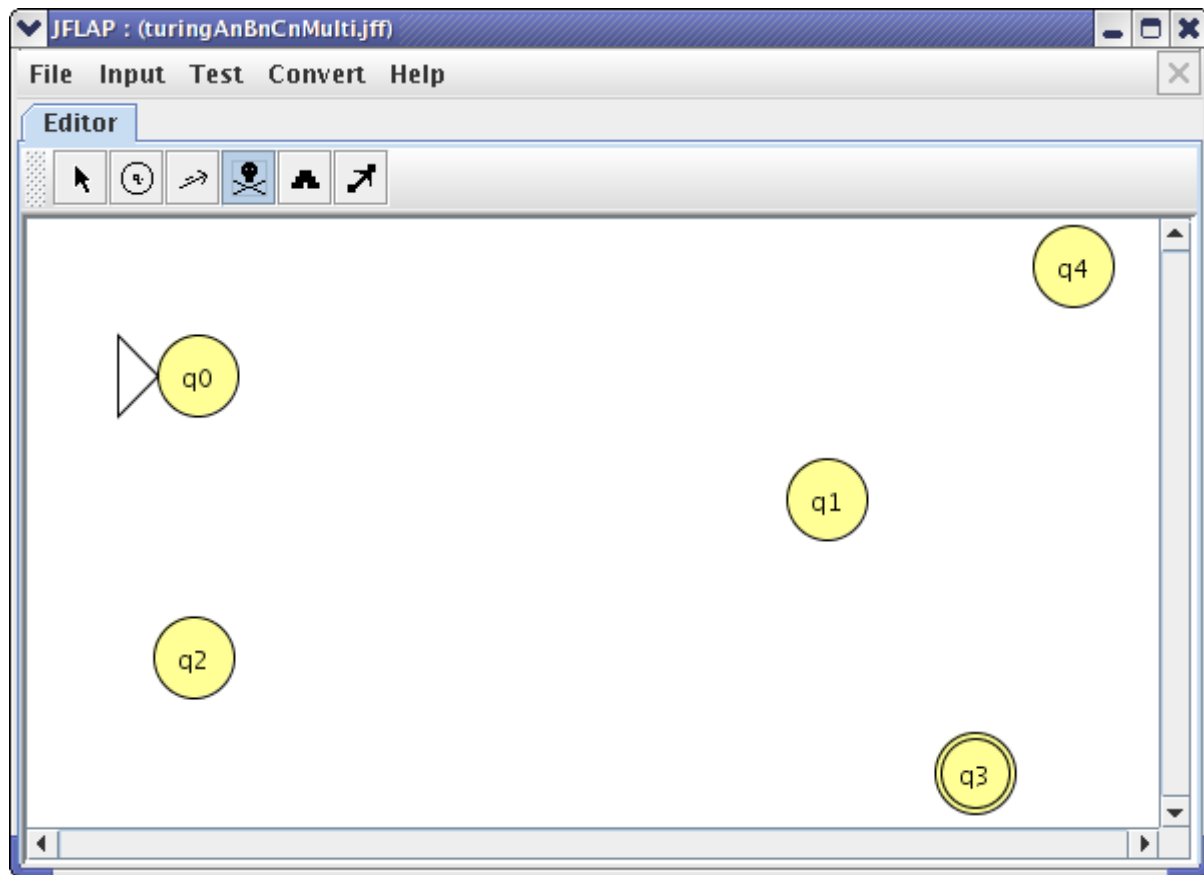
### How to Create a Multi-Tape Turing Machine

To start a new multi-tape Turing machine, start JFLAP and click the **Multi-Tape Turing Machine** option from the menu, as shown below:



A pop-up window will come up asking how many tapes to implement, 2 is the default, but from the menu given, we can choose a number in the range of 2 to 5. While we could definitely do this problem with 1 or 2 tapes, choose 3 tapes, because it allows for us to implement an automaton that is very easy to understand visually. After choosing 3 tapes, one will see a screen that is identical to that for one-tape Turing machines. However, there are a few different features that we will soon encounter.

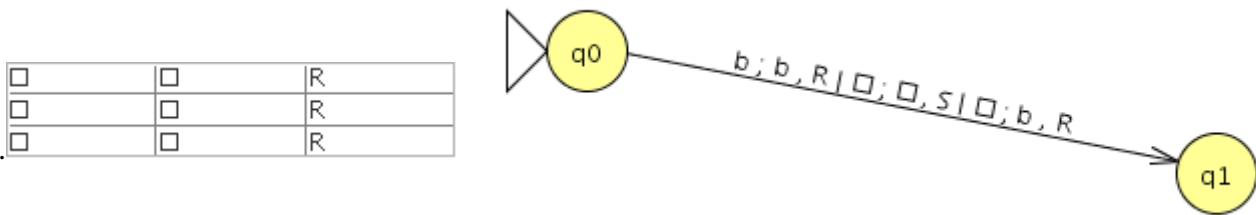
On the blank screen, add 5 states to the screen, so that the screen resembles the one below.



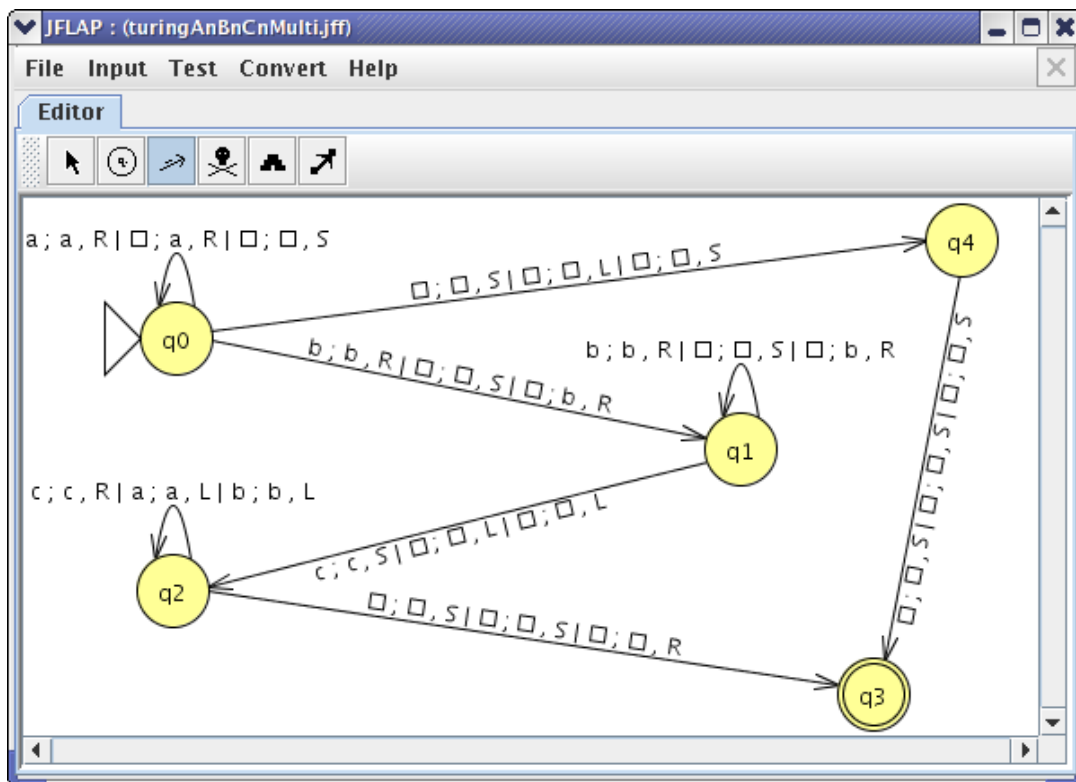
Now its time to add the transitions. Attempt to add a transition between the states q0 and q1. You'll notice that, instead of just one tape that you have to define values for, you must define values for three tapes. Go ahead and enter in (b; b, R) for the first tape, ( $\square$ ;  $\square$ , S) for the second, and  $\square$ ; b, R) for the third. This transition will check to see if there is a 'b' under the head of the first tape, and if there are blanks under the heads of the other two tapes. If this is true, then it will add a 'b' to the third tape, move the heads of the first and third tapes to the right, and not move the head of the second tape.

### Three tapes

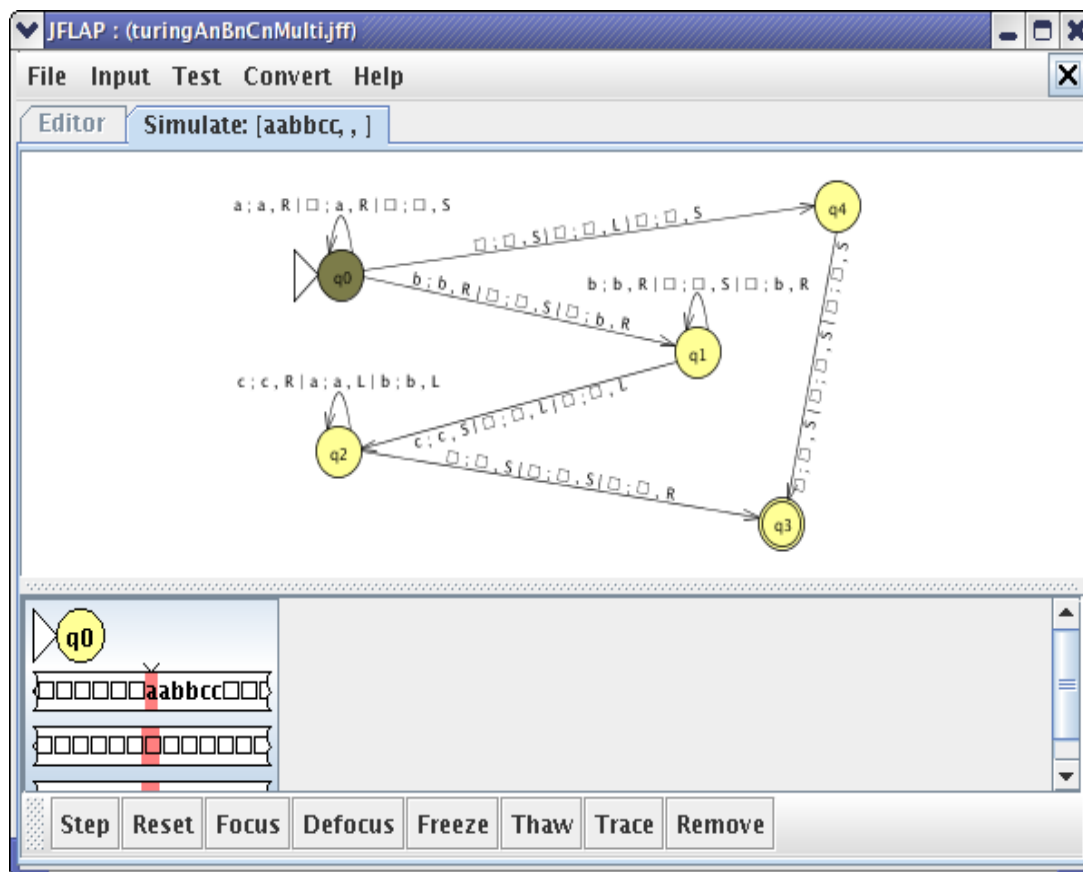
### The first transition



Let's finish up the transitions. Add the transitions in your screen below to your Turing machine. If you would rather not add every transition directly and would prefer to load the file of the screen below, it is available at [turingAnBnCnMulti.jff](#).

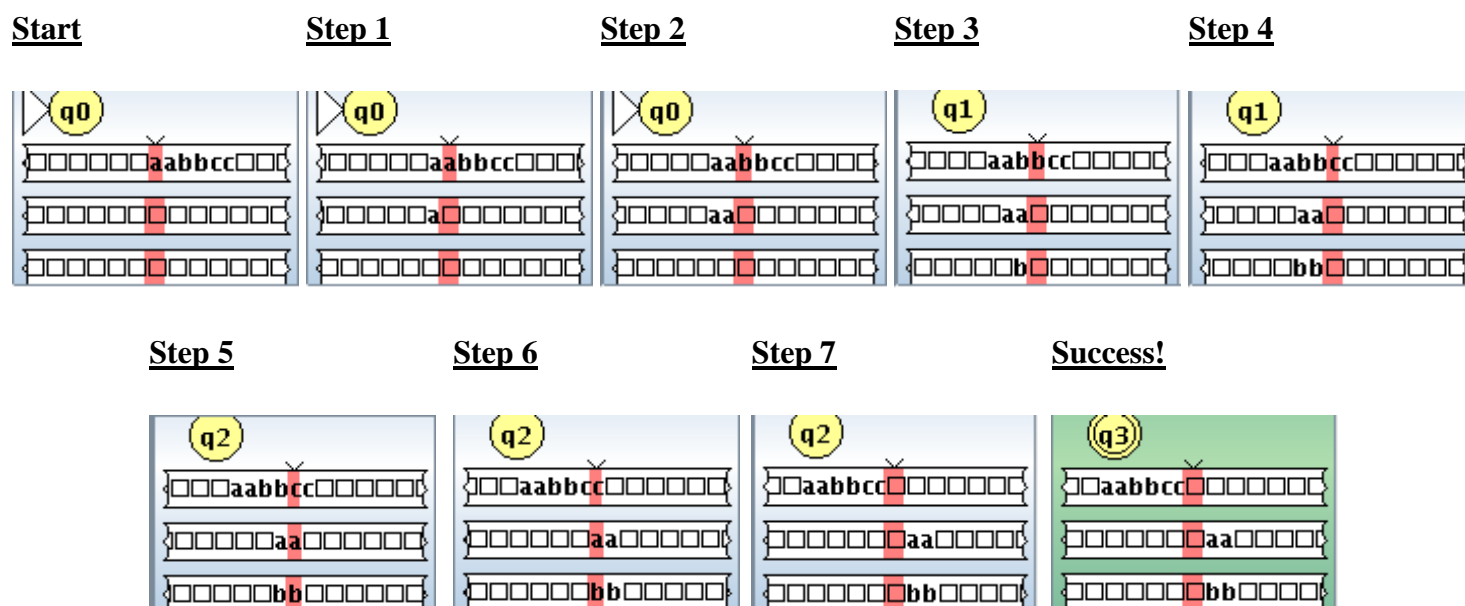


Now, let's try out our new multi-tape Turing machine. We could do a “Fast Run”, but since we haven't done a “Step” option yet for Turing machines, let's try that. It is very similar to the “Step with Closure” option for Finite Automata. First, click on “Input” and then “Step”, and when it prompts for the values for the three tapes, enter the string “aabbcc” on the first tape (representing  $a^2b^2c^2$ ) and nothing on the next two tapes. We now see the screen below...



The box in the lower left is different from that which appears for the Finite Automata's “Step with Closure” option. In this example, each tape is present, and a red highlighted area represents the current position of the tape's head. A similar screen will appear for any number of Turing machine tapes, with one tape for single-tape machines and five for five tape machines (all viewable with the scroll bar at the lower right). At present, the third tape is partially obscured, so we should scroll down until it is more visible or resize the window. Which ever option you choose, make sure you can see all three tapes.

As we press step repeatedly, the following are the steps through which the simulation runs:





We now see firsthand how the algorithm works and how it is implemented in action. First, the first tape reads all the 'a's and copies them onto the second tape. Then, it reads all the 'b's and copies them onto the third tape. Finally, all three tapes check to see whether the number of 'a's, 'b's, and 'c's are equal, the first tape checking the 'c's, the second the 'a's, and the third the 'b's. We might wonder where state “q4” comes into this, but transitions leading to and from it simply check where  $n = 0$ , which cannot be served by the other transitions.

## Building Blocks

---

### Updates in Version 7.0

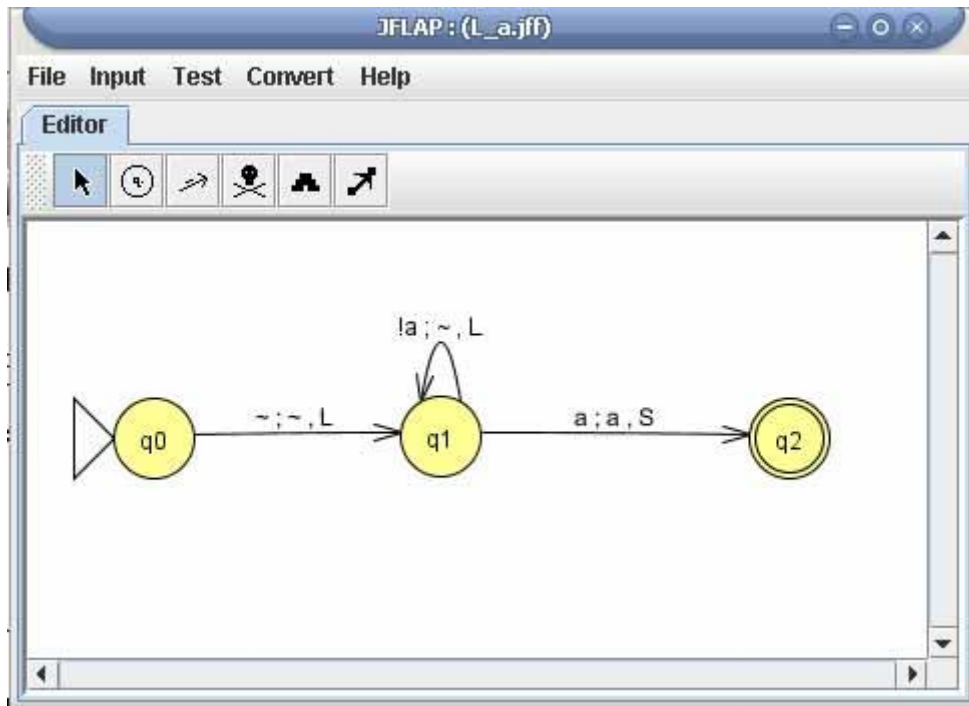
In 7.0, the distinction between ordinary states and building blocks is completely eliminated. In order to make that clear visually, all Turing Machine states are now circular, whether they are building blocks or not. Every ordinary state can become a building block, by simply right clicking on it, and clicking Edit Block.

Turing machines within building blocks must have an initial state, although final states will be ignored. JFLAP begins to look for transitions out of a building block when the TM within the block halts. Please see the [Preferences](#) page of the tutorial for information on new preferences for Turing machines.

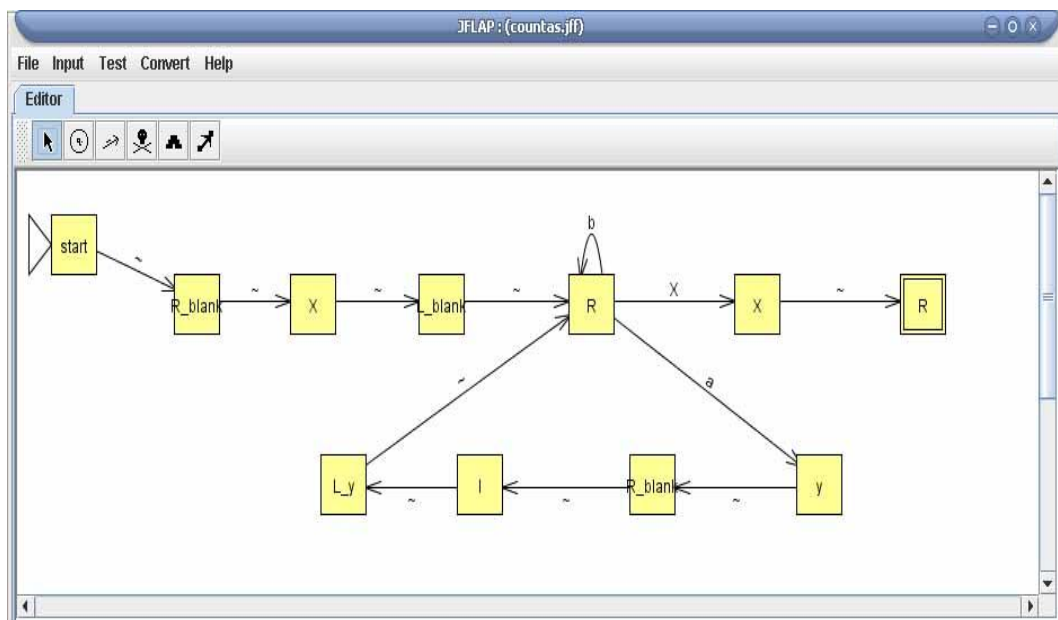
---

All files that are used in this tutorial, both the machines demonstrated and the library of building blocks used to build them, are available in [buildingblocks.zip](#).

Building Blocks are small Turing machines you can build or get from a library of prebuilt machines. We have provided a few simple blocks to get you started, these are: "Move Left Until a", "Move Right Until a", "Move Left Until Not a", "Move Right Until Not a", and similar blocks for b's and blanks. The internals of the Building Block "Move Left Until a" is shown below.



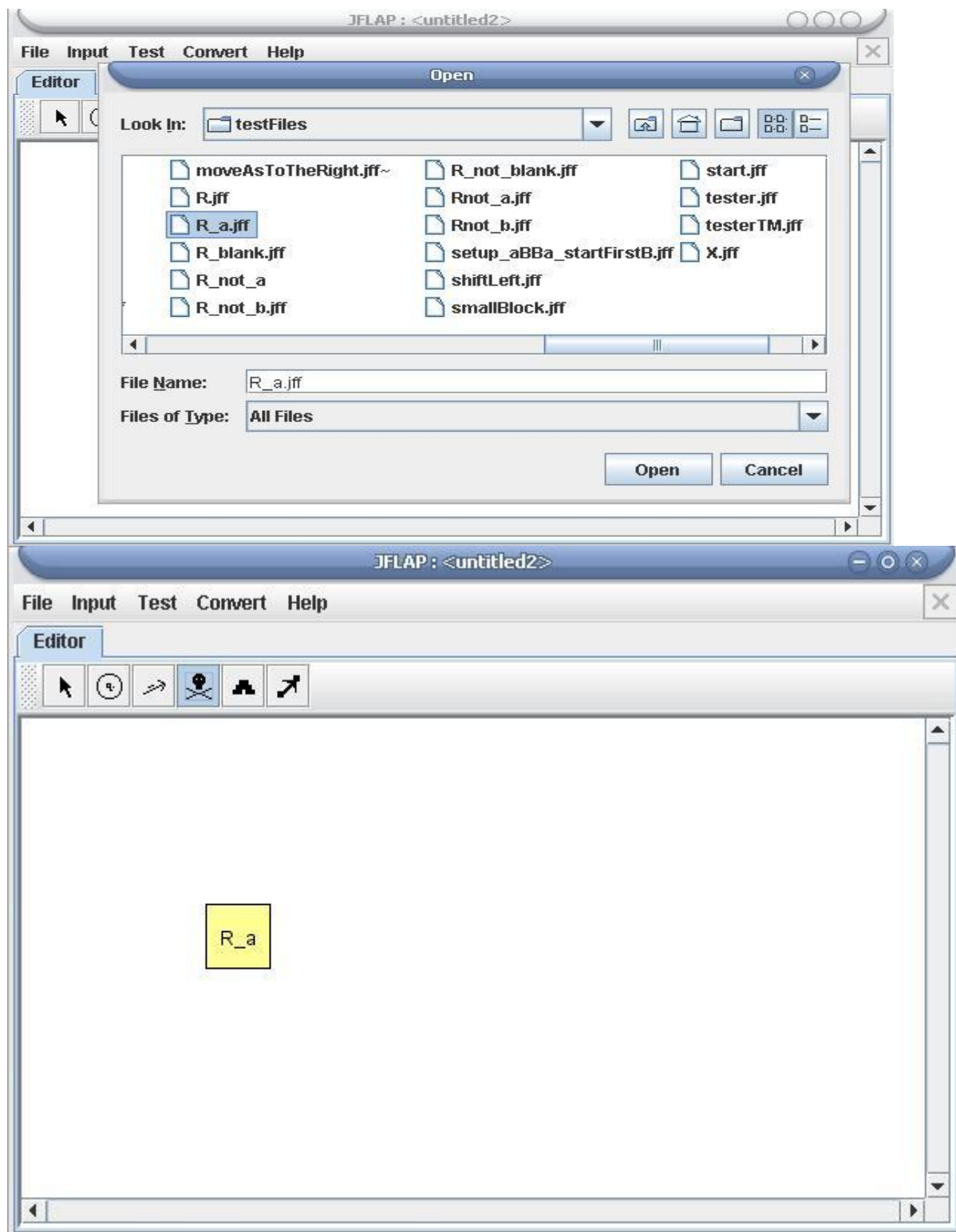
Building blocks are meant to be used to assemble larger Turing machines quickly and easily. Any Turing machine built in JFLAP can be used as a building block by selecting the Building Block Tool (or pressing b), clicking in the editor pane, and choosing the file to use as a building block. Below is a Turing machine made out of building blocks that changes all “a”s to “y”s, adds an “X” character to the end of the input, and outputs after the “X” a number of “I”s that correspond to the number of “a”s. Thus, it serves to count the number of “a”s. For example  $f(ababb) = ybybbXII$  and  $f(baabaa) = byybyyXIII$ . The alphabet is  $\{a, b\}$ .



To add a building block to a file use the Building Block Creator tool, highlighted in the image below.



Click on an empty part of the editor. A file open dialogue appears for you to choose what file to import as a block. When you find the file and press open it will appear as a square with the name of the file inside it.



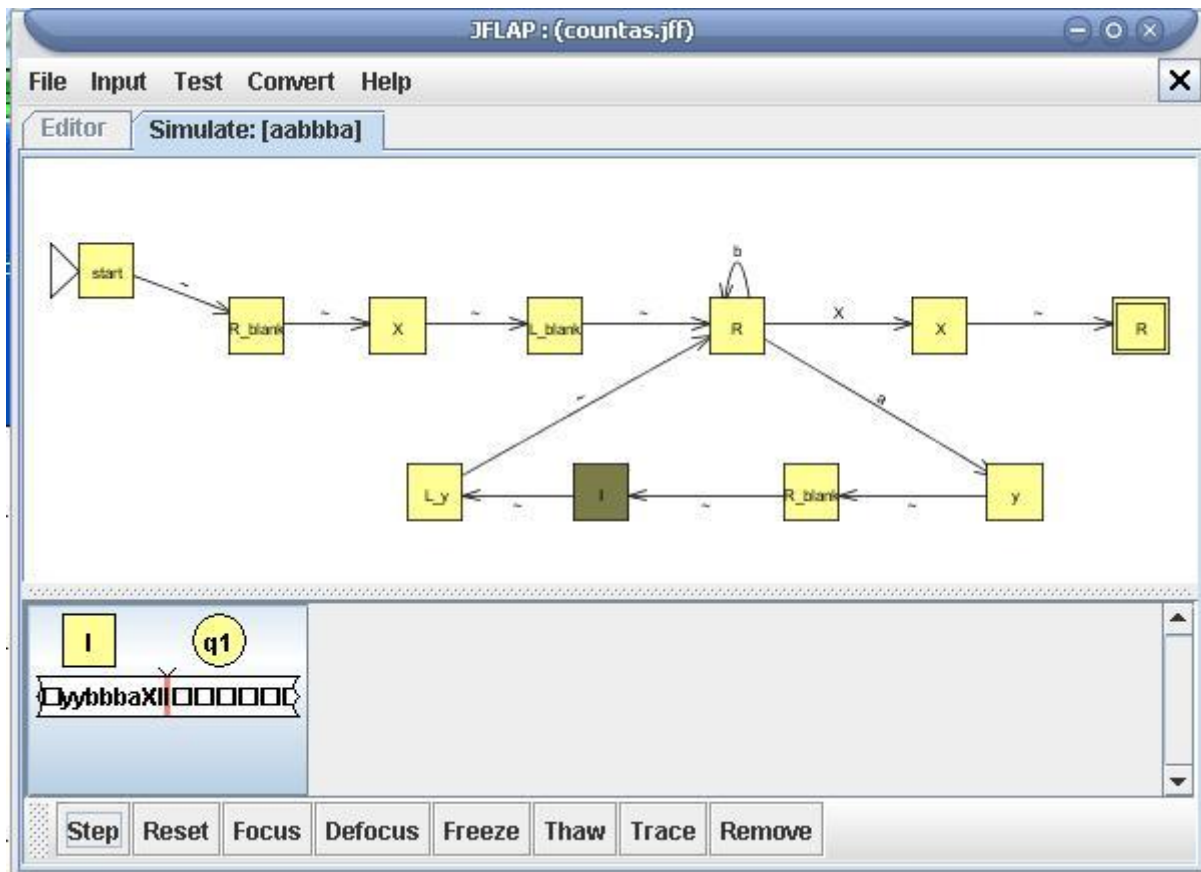
## Running Turing Machines With Building Blocks

The default view of a Turing machine while running is to view at the highest level, which only shows what block the configurations are in. To see the states within the block, you can select a configuration and then choose the Focus button. From that point until Defocus is chosen JFLAP

will follow the configuration through the lowest level. displaying the state it is in inside the block. The focused configuration is colored yellow in the configuration pane.

To speed up stepping through the Turing machine when you don't want this kind of detail we have provided a new type of step called Step By Building Block. When chosen from the Input menu it will process an entire building block every time the Step button is clicked. Specifically, every time the step button is pressed JFLAP will continue running each configuration until it exits the block.

Here the machine that counts a's from above is shown in the middle of being tested.



### Editing Building Blocks:

Once a block, say "smallBlock.jff" has been added to your Turing machine, it is stored inside that turing machine's file which we'll call "largeMachine.jff", so making changes to the original file, "smallBlock.jff" will not change the building block inside "largeMachine.jff". To change the block select the Attribute Editor by pressing the letter a, right click on the block, and choose Edit Block. This opens a new tab in which you can make changes to the block as with any other Turing machine. When you are done making changes open the File menu and choose Dismiss Tab. This will save the changes. Multiple blocks from the same file may be added to a turing machine, but changes to one block will not affect the other blocks. When editing blocks you should be careful to update the name of the block to reflect the new changes as it can be easy to forget what you did making debugging difficult.

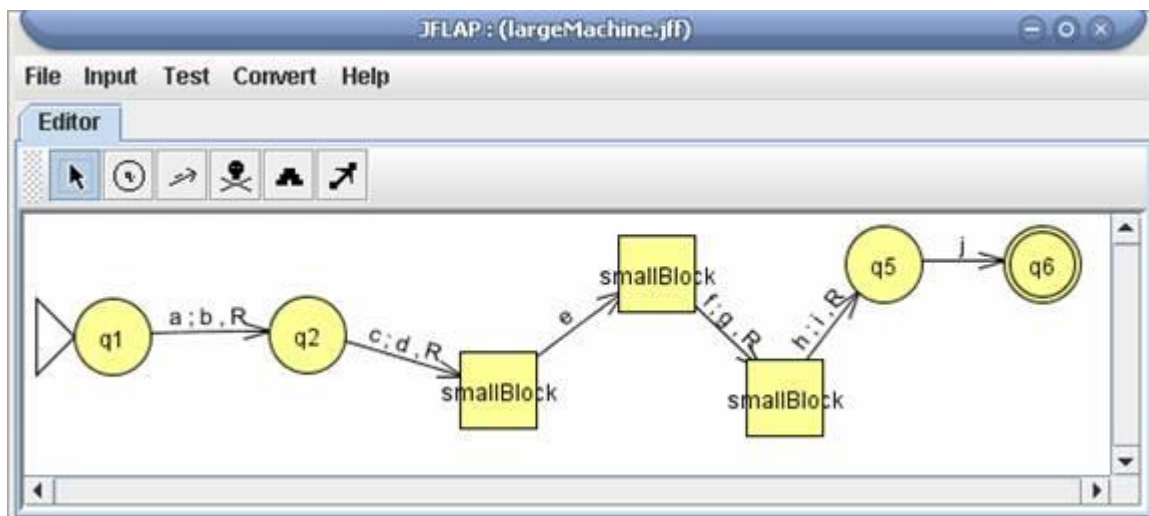
### Mixing Blocks and States:

Building Blocks can be treated like States in that transitions can be added to them just like States. However, there is also a Block Transition Creator tool that can be used instead, highlighted in the image below.



This type of transition allows input of one letter. When running the Turing machine, if it reaches the final state of the first block, and the head is pointing to the same letter as in the transition, it will move into the initial state of the second block. The tape head will not move, and no letter will be written to the tape.

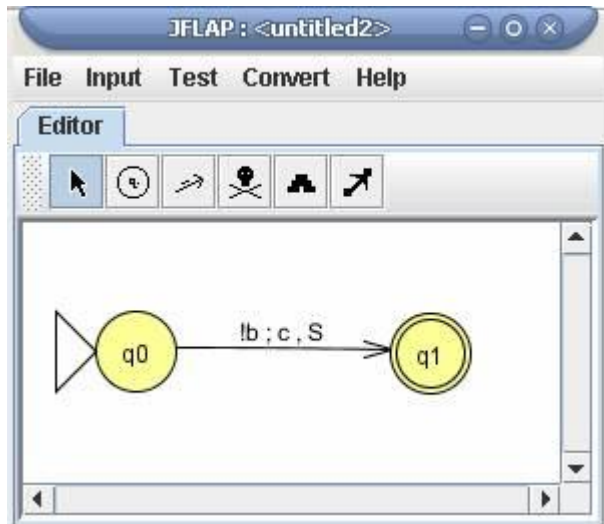
As you can see from the image below, any kind of transition can be added between any combination of blocks and states.



### New Transition Syntax

#### Not X:

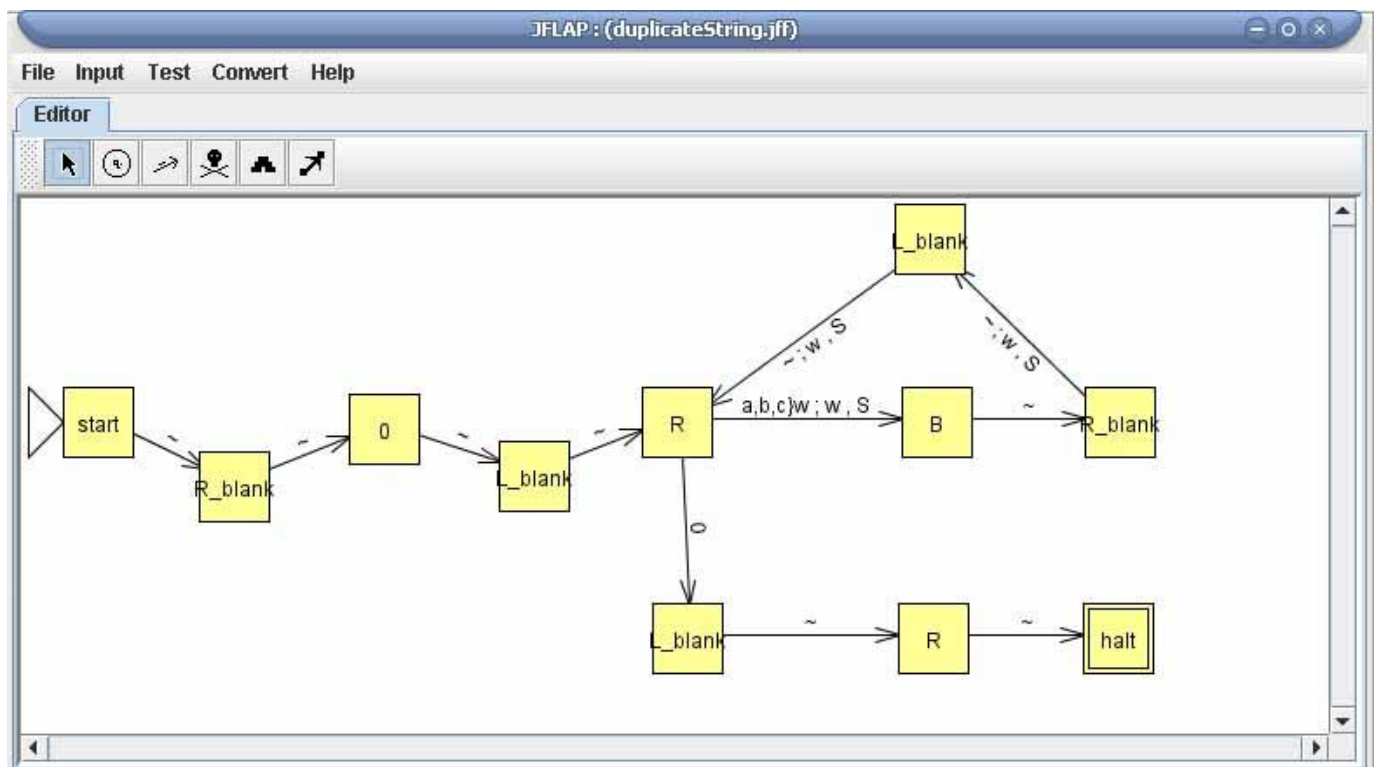
This allows transitions that read in any letter other than the ones specified, instead of having to create one for each letter. For instance if you wish to transition from state one to state two after reading all letters except b, writing the letter c always, you could use the syntax read !b, write c, S. As shown in the machine below.



### Variable Assignment :

To use this feature create a regular transition between two states or blocks, we'll call them A and B, and in the first field type the letters you wish to assign to the variable. For example “a,b,c}w” would mean that when the Turing machine got state A if the head was on either a, b, or c the letter at the head would from then on be synonymous with w. This means that later transitions that say read w or write w actually mean read the letter that was at the head or write that letter. An example machine is shown below. The first transition says to store the letter at the head as w and move the head right. The next transition says if the new letter at the head is the same as w, write b and move right again. The input is only accepted when two of the same letters are in a row, aa, bb, or cc. The output if used as a transducer is ab, bb, or cb.

Below is a Turing machine that duplicates the input string separated by 0 over the alphabet {a, b}. Input of aababacc for example gives aababacc0aababacc.



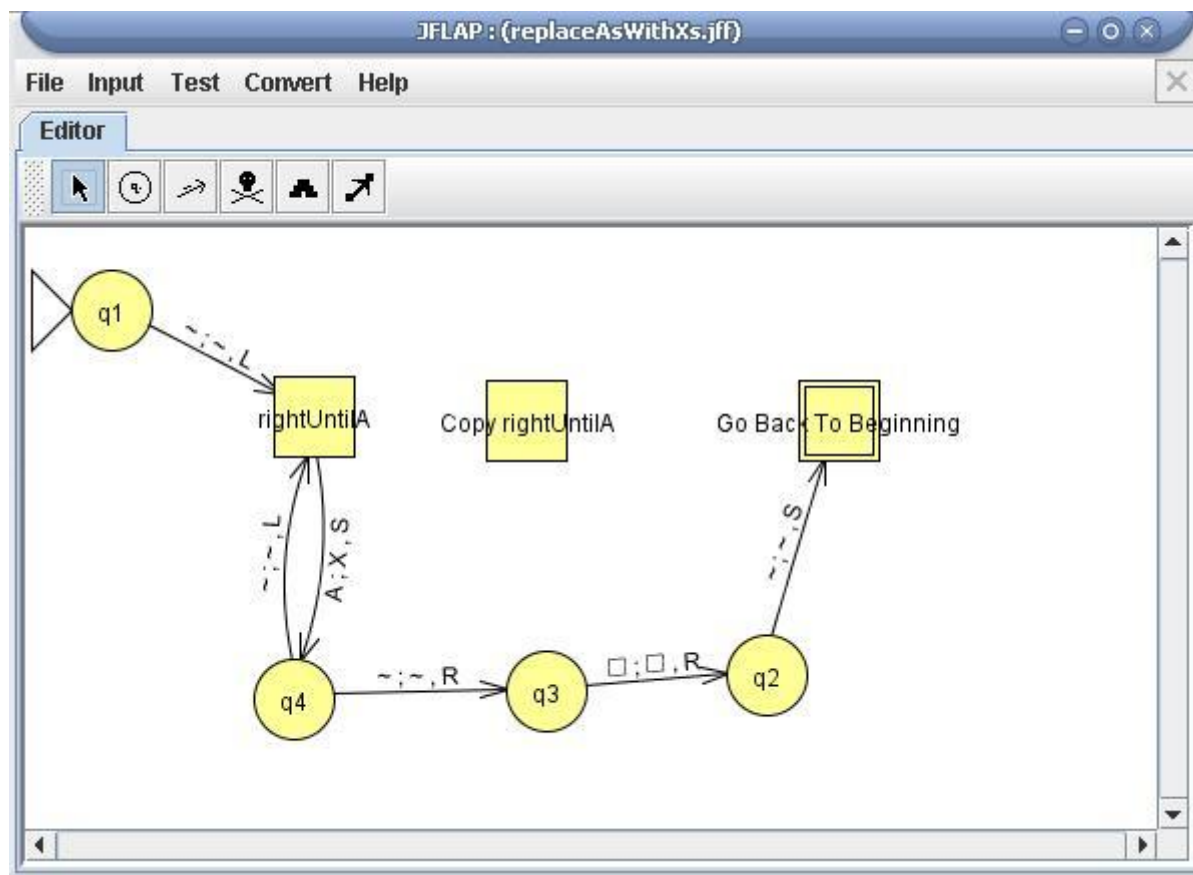
When the machine starts it moves right until it finds a blank, rights a 0, moves left until it finds a blank, moves one right so that the head is on the first letter of the input string. Out of this block there are two transitions; we'll focus on the one that says a,b,c}w, w, S. This states that if the letter pointed to by the head is a, b, or c that letter will be assigned to the variable w. We then write w to the tape, which is the same thing that was just read. In the next step we write a blank over what was just written, move right until the next blank, and write w over it. We then move left until the first blank and rewrite w over the blank. We are now back in the fifth block from start, labeled R and ready to continue duplicating the string or branch to the final state if everything has been duplicated.

## Building Block Tools

### Duplicate Block:

If you need a copy of a block, right click on it and select Duplicate Block. By default it will be named Copy of x, where x is the name of the original block, and will appear directly under the original. As with blocks loaded from the same files, changes to one will not affect the other as they are not linked.

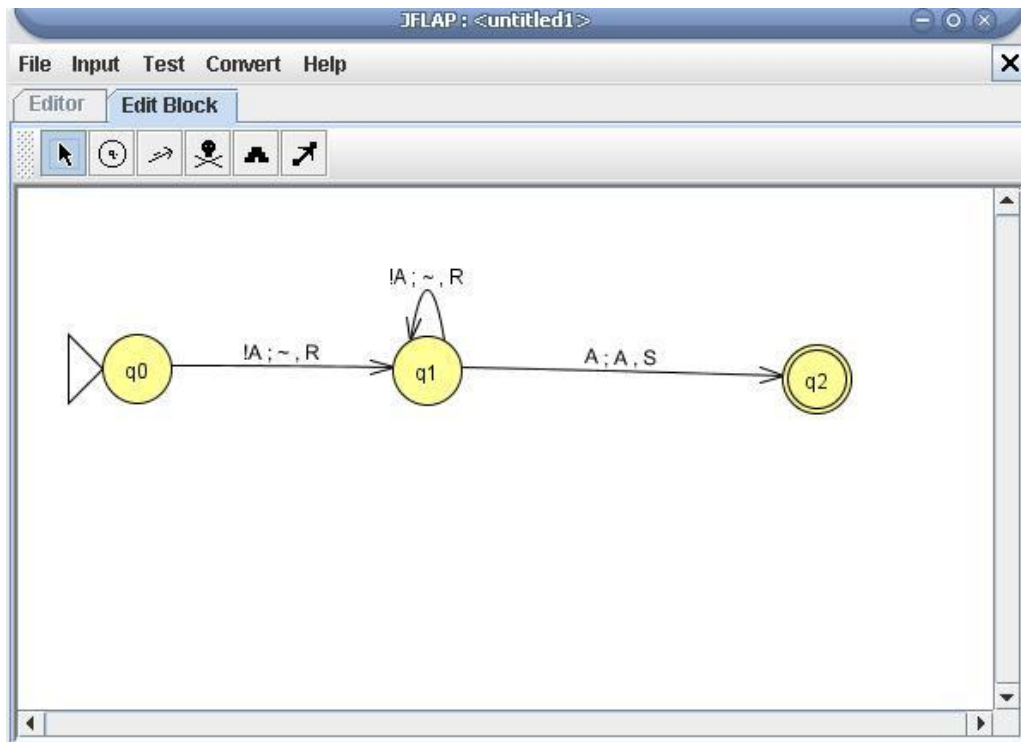
This tool is especially useful for situations where you have loaded multiple blocks from the same file before discovering an error in the original file. Now all you have to do is edit one block to fix it, erase all the other blocks that were loaded from the file, and then replace them by duplicating the fixed block.



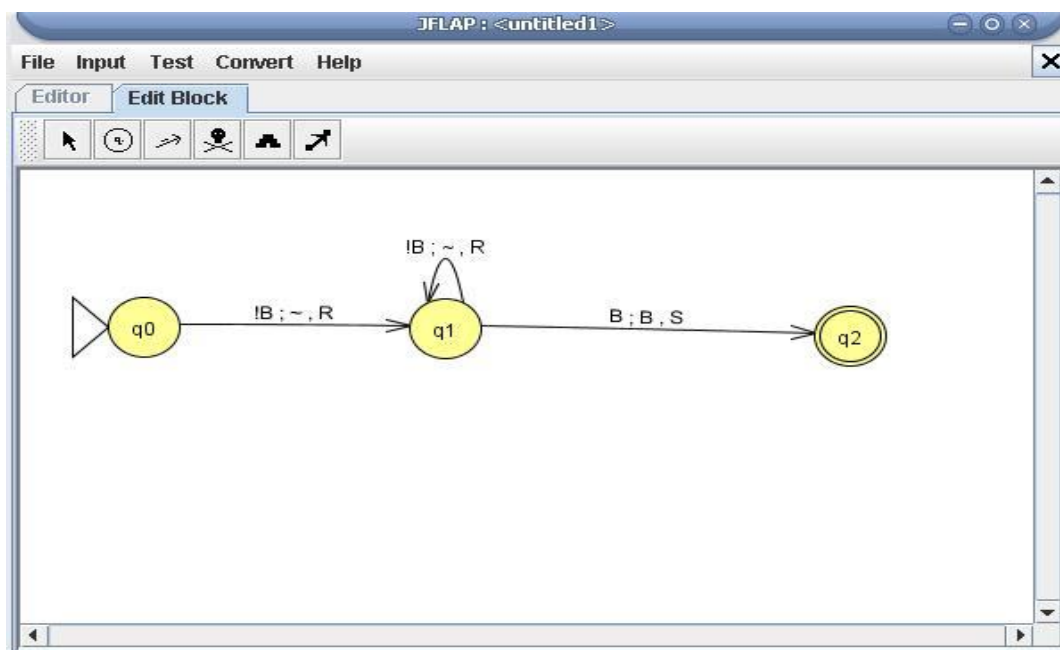
### Replace Character Within Block



To replace all occurrences of a letter within a block, right click on the block and choose Replace Symbol. Type the character you want to replace and press enter, then type the character you want to take it's place and press enter. The character will be replaced in all transitions inside the block including all transitions inside all sub-blocks. The picture above shows a file that changes all A's in the input string to X's. Here is the inside of the block rightUntilA.



To change the file to one that changes all B's in the input string to X's instead, use Replace Character Within Block. Type A, press Enter, type B, press Enter. Now the inside of rightUntilA looks like this.



Be careful to change the name of the block to rightUntilB to avoid confusion.



## 14. Converting Turing Machine to an Unrestricted Grammar

### Contents

Introduction

Converting to an Unrestricted Grammar

Export and Parse

### Introduction

Any Turing machine can be converted to an unrestricted grammar.

JFLAP defines an unrestricted grammar as a grammar that is similar to a context-free grammar (CFG), except that the left side of a production may contain any nonempty string of terminals and variables, rather than just a single variable. In an unrestricted grammar, the left side of a production is matched, which may be multiple symbols, and replaced by the corresponding right hand side.

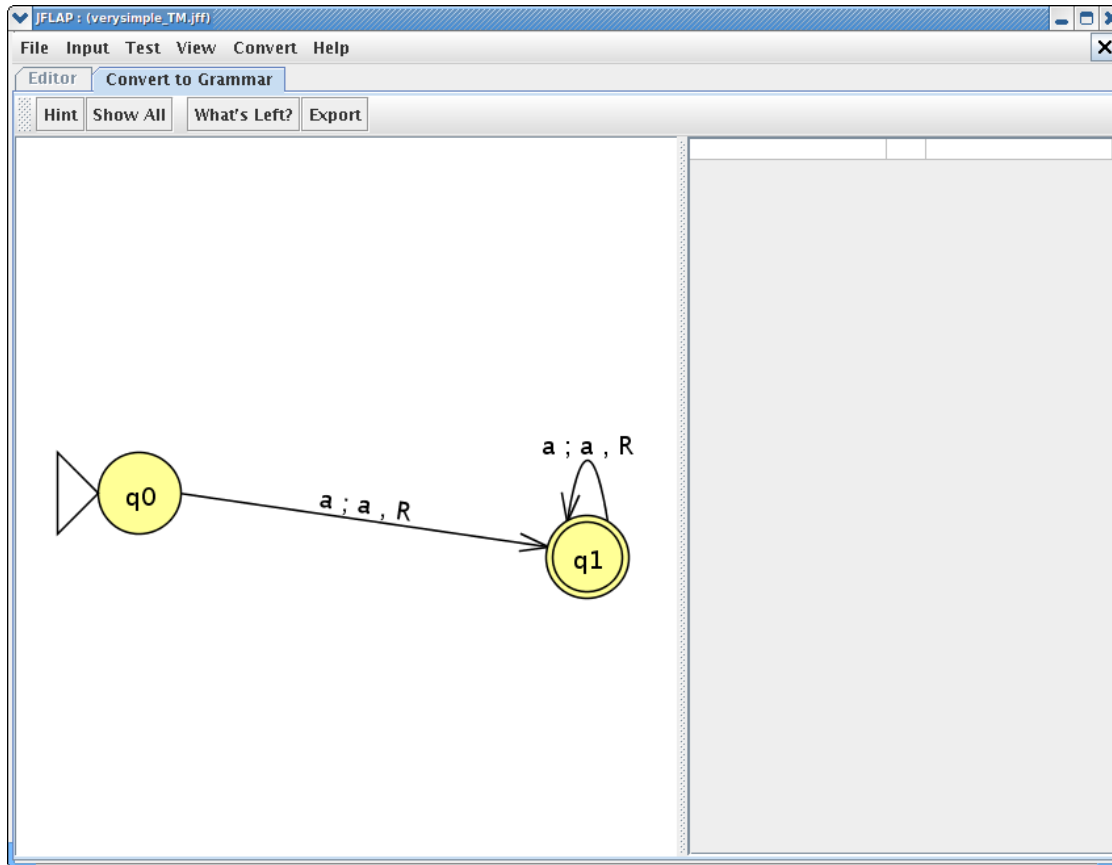
This conversion implements the algorithm in the book “An Introduction to Formal Languages and Automata 4<sup>th</sup> Edition” by Peter Linz. The algorithm can be founded on page 283-285.

There are 3 major steps that the algorithm follows in order to convert the Turing machine to an unrestricted grammar. The first step is applying the basic rules from the start variable. It allows the grammar to generate an encoded version of any string  $q_0w$  with an arbitrary number of leading and trailing blanks. The second step is applying generating the production for each transition of Turing machine. The last step is applying additional rules to our productions if we enter one of final states of TM, so we can derive terminals.

### Converting to an Unrestricted Grammar

We will now convert the TM into an unrestricted grammar. First, open up the Turing Machine window and load the file [verysimple\\_TM.jff](#).

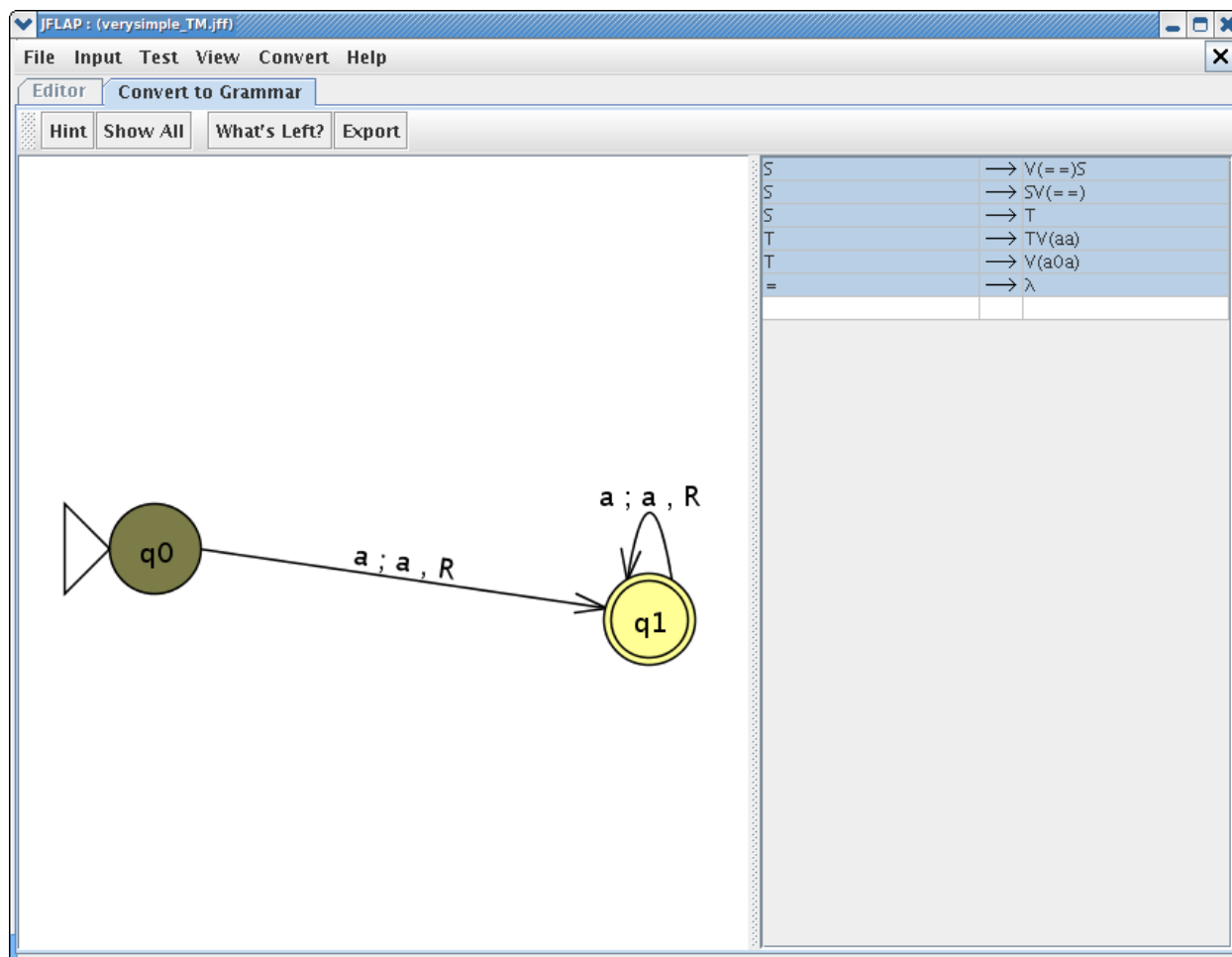
Click on the **Convert** → **Convert to Unrestricted Grammar** menu option, and the following screen should appear :



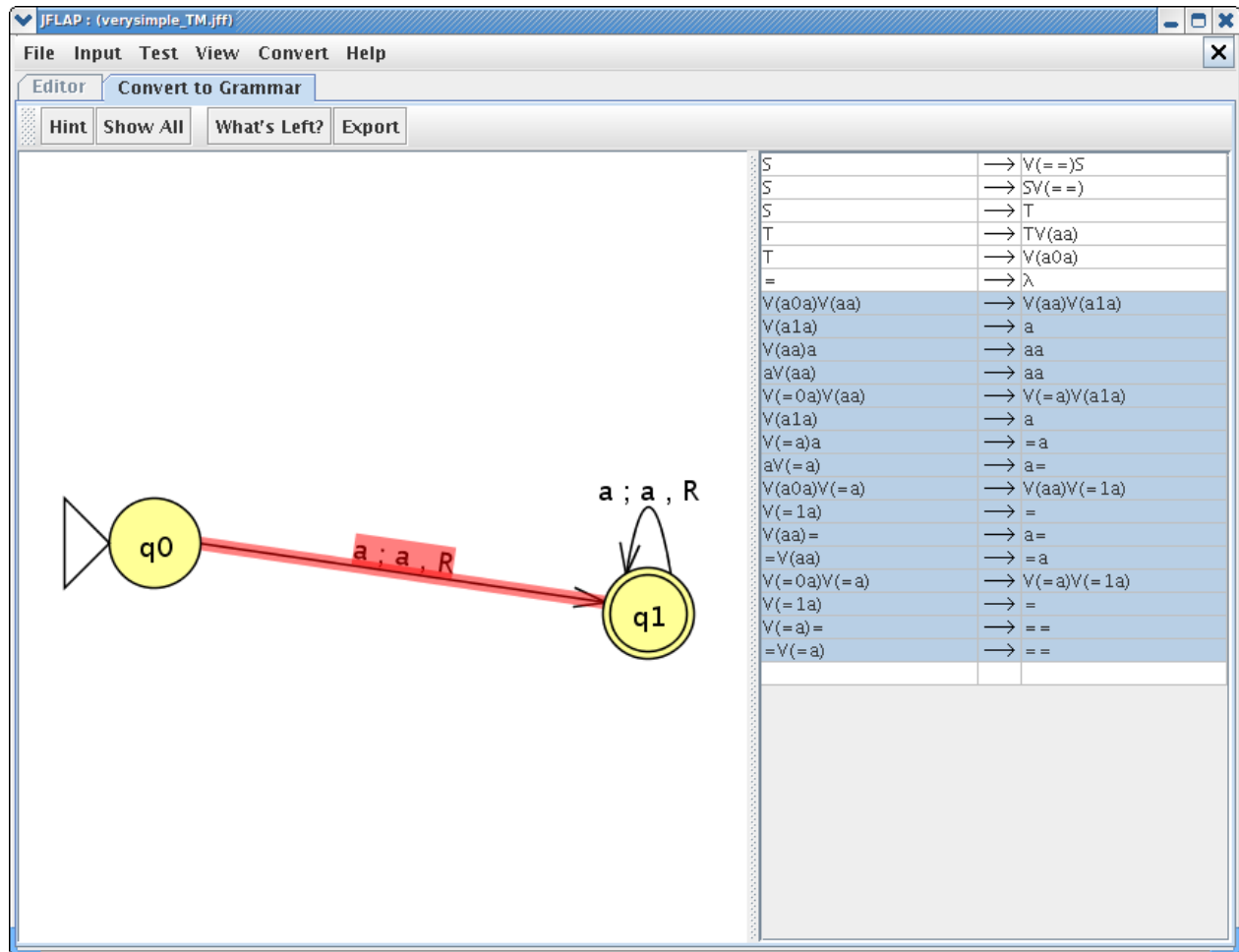
This is a very simple Turing machine that accepts any language that has one or more “a”s. Just like other conversion windows, you can click the **What's Left?** button to view what states or transitions remain to be converted.

Now, let's produce our first productions for our new grammar (the first step of algorithm). By default, there are going to be 3 productions that are going to be produced for every TM that is going to be converted. These three rules are

“ $S \rightarrow V(==)S$ ,  $S \rightarrow SV(==)$ ,  $S \rightarrow T$ ”, where  $S$  is the start variable. In addition there are going to be 2 additional rules “ $T \rightarrow TV(aa)$ ,  $T \rightarrow V(a0a)$ ”, in which “ $a$ ” refers to for all readable strings in the tape. The result of productions generated by the first step can be viewed by clicking initial state  $q0$ . Note that “ $=$ ” refers to the blank tape symbol in the Turing machine. After clicking the state  $q0$ , your window would look like this :



Now, let's move on to the second step of algorithm and examine our first transition from  $q_0$  to  $q_1$ . The transition is  $(q_0, a) \Rightarrow (q_1, a, R)$ . By following the algorithm, we are going to place production  $V(x0a)V(yz) \rightarrow V(xa)V(y1z)$ , where  $x, y$  is all elements of input alphabet+blank symbol and  $z$  is all elements of the tape alphabet. Since we only have "a" in our input alphabet our  $x$  and  $y$  is going to be "a". "z" can be either "a" or "=" (blank). Also, since  $q_1$  is final state, we have to apply the last step of algorithm. We apply the special rule  $V(a1a) \rightarrow a$  and  $yV(xz) \rightarrow a$ , where  $x$  and  $y$  are all the input alphabet + blank symbol (=) and  $z$  is all of our tape alphabets. By doing so, we can generate terminals on the right hand side of productions. After applying all of these rules, our grammar window would look like :



After either clicking our last transition in  $q_1$  or using **Show All** button, we can finish our conversion.

JFLAP : (verysimple\_TM.jff)

File Input Test View Convert Help

Editor Convert to Grammar

Hint Show All What's Left? Export

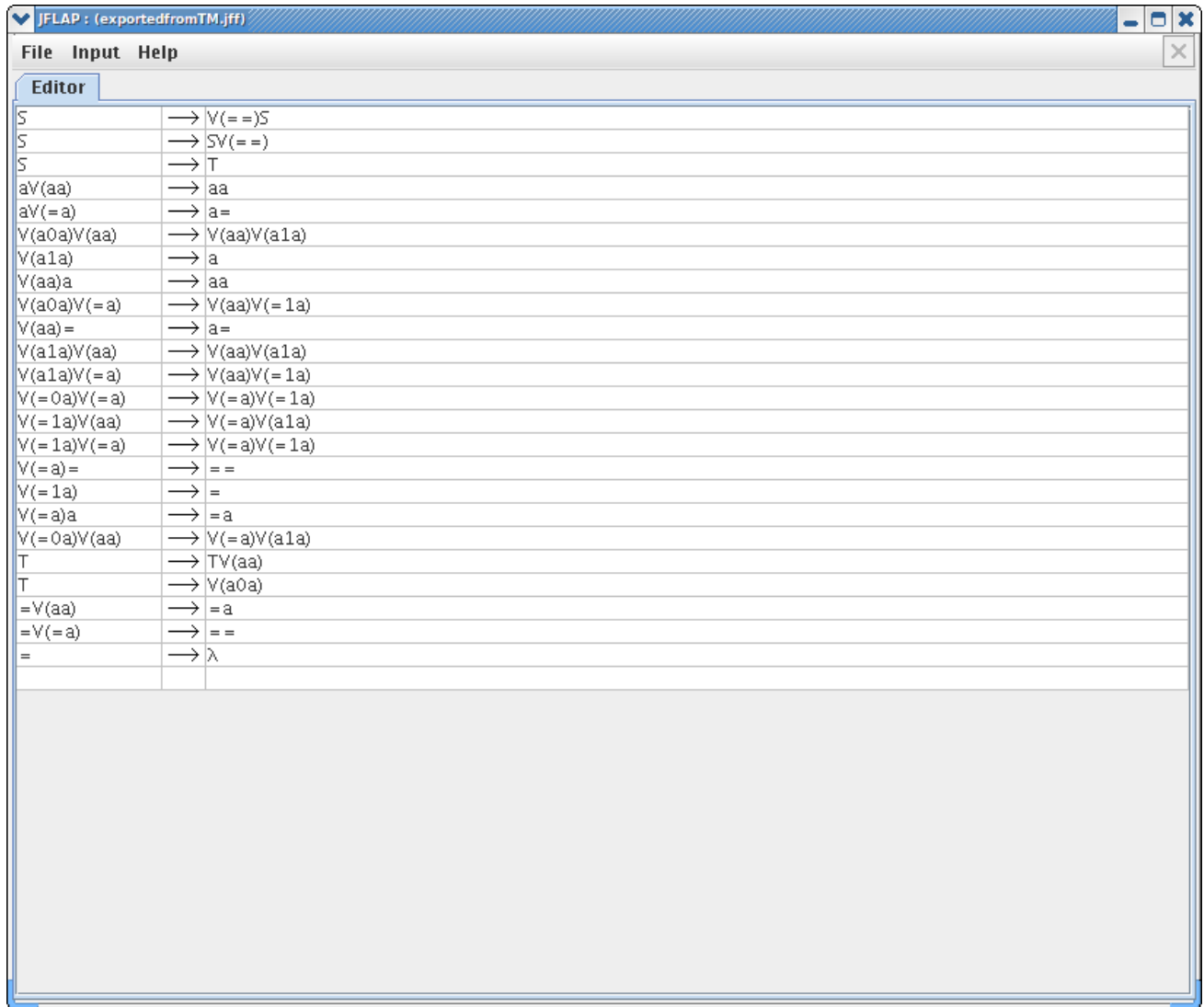
```

graph LR
    start(( )) --> q0((q0))
    q0 -- "a ; a, R" --> q1(((q1)))
    q1 -- "a ; a, R" --> q1
  
```

S	→	V(=)S
S	→	SV(=)
S	→	T
T	→	TV(aa)
T	→	V(a0a)
=	→	λ
V(a0a)V(aa)	→	V(aa)V(a1a)
V(a1a)	→	a
V(aa)a	→	aa
aV(aa)	→	aa
V(=0a)V(=a)	→	V(=a)V(a1a)
V(a1a)	→	a
V(=a)a	→	=a
aV(=a)	→	a=
V(a0a)V(=a)	→	V(aa)V(=1a)
V(=1a)	→	=
V(aa)=	→	a=
=V(aa)	→	=a
V(=0a)V(=a)	→	V(=a)V(=1a)
V(=1a)	→	=
V(=a)=	→	=
=V(=a)	→	=
V(a1a)V(aa)	→	V(aa)V(a1a)
V(a1a)	→	a
V(aa)a	→	aa
aV(aa)	→	aa
V(=1a)V(aa)	→	V(=a)V(a1a)
V(a1a)	→	a
V(=a)a	→	=a
aV(=a)	→	a=
V(a1a)V(=a)	→	V(aa)V(=1a)
V(=1a)	→	=
V(aa)=	→	a=
=V(aa)	→	=a

## Export and Parse

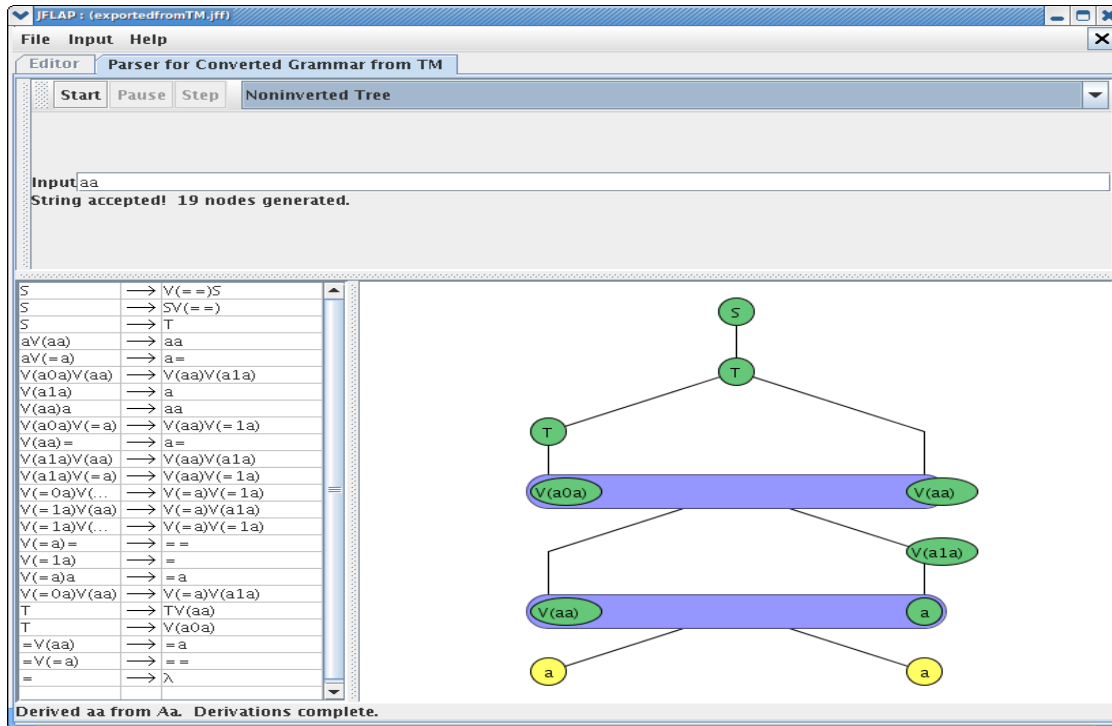
Let's now export this converted grammar. You can click on **Export** button to export this grammar. After exporting the grammar, your window would look like this :



Now let's try to parse this unrestricted grammar, note that  $V(aa)$  is actually refers to one variable. Click on **Input** and **Parser for Converted Grammar from TM**. You will see very similar grammar parse pane from brute force parsing. In this parser, all the variables such as  $V(aa)$  is trimmed into regular variable like "A" or "B", so the parsing can occur. This trimming procedure is done internally in the parser. In the input, type in string "aa". You can see that the this string is accepted by the language as expected. Click on **STEP** buttons several times to see how we derive this string. You can either view the derivation in Noninverted tree or Derivation table. Your result should look like one of these windows.

**NOTE:** The JFLAP Parser recognizes this special notation in the grammar. It converts variables such as  $V(aa)$  into normal variables like "A" in the background to facilitate the parsing.

**WARNING:** This only works with very small examples since the conversion is exponential.



JFLAP : (exportedfromTM.jff)

File Input Help

Editor Parser for Converted Grammar from TM

Start Pause Step Derivation Table

Input: aa  
String accepted! 19 nodes generated.

Production	Derivation
S → T	S
T → TV(aa)	T
T → V(a0a)	TV(aa)
V(a0a)V(aa) → V(aa)V(a1a)	V(a0a)V(aa)
V(a1a) → a	V(aa)V(a1a)
V(aa)a → aa	V(aa)a
V(aa)a → aa	aa

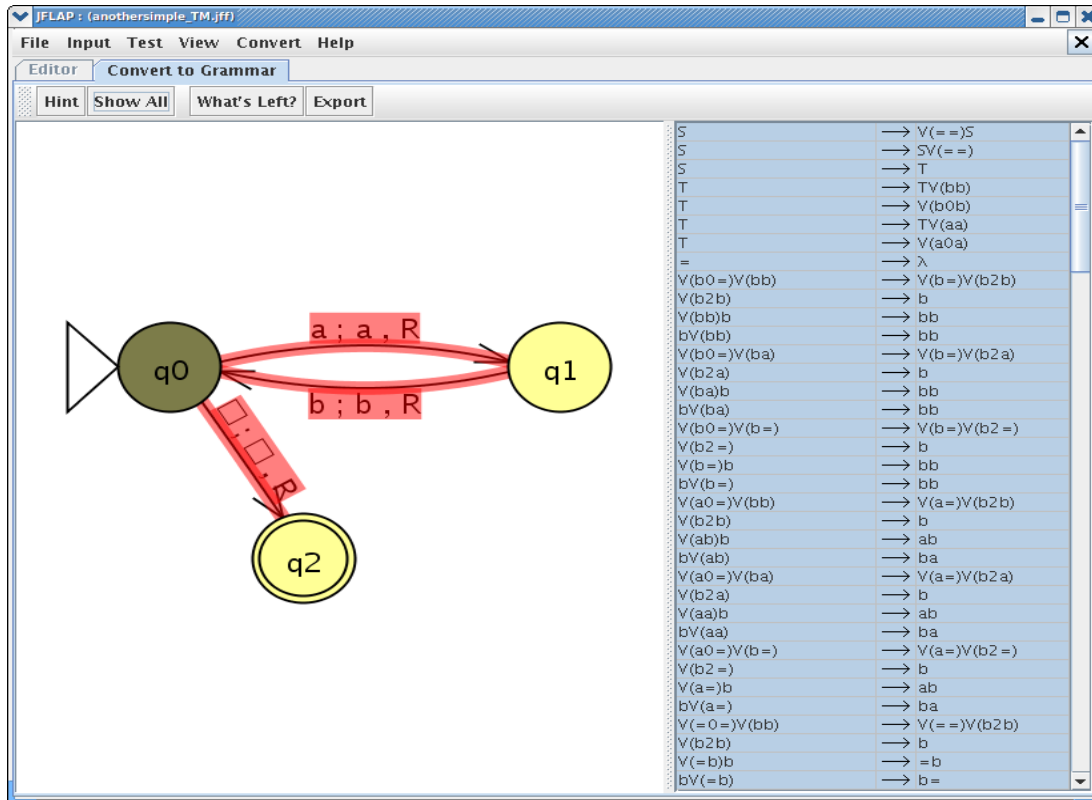
Derived aa from Aa. Derivations complete.

Now let's try to input "aaaaaaaa" (8 a's). The parser quickly accepts the language, but note that there are 271 nodes generated for such a simple string. This reveals that the parser is going to have really hard time parsing other complicated grammar such as  $a^n b^n c^n$ . As Turing machine gets more complicated, the parsing of converted grammar would get more difficult and almost

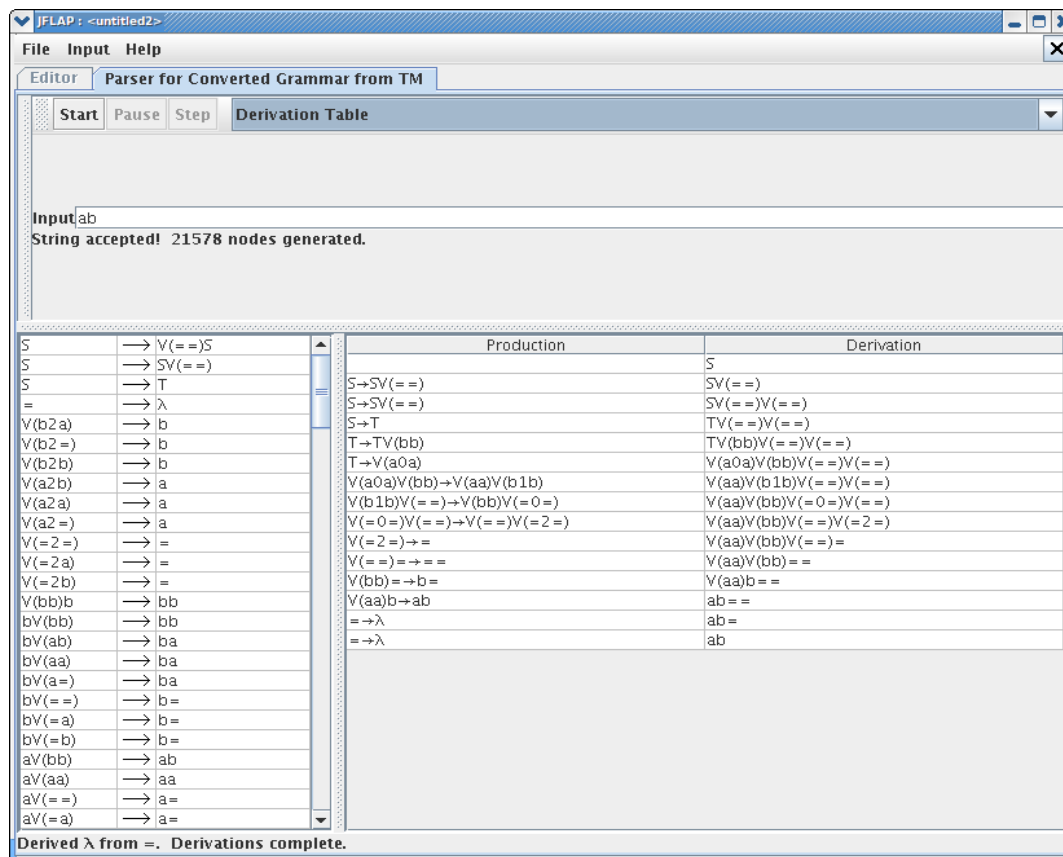
impossible. Although the converted grammar might not be useful in actually checking the string, it proves that TM can be converted to unrestricted grammar.

**NOTE:** You can also use user control parser to derive the string manually.

Another example of converting TM to unrestricted grammar is shown in below windows. You can load the file [another simple TM.jff](#) to load this example. Note that it generated 21578 nodes to just to accept simple string “ab”.







## 15.Convert CFG to PDA (LL)

### Contents

#### Definition

#### How to Convert CFG to PDA (LL)

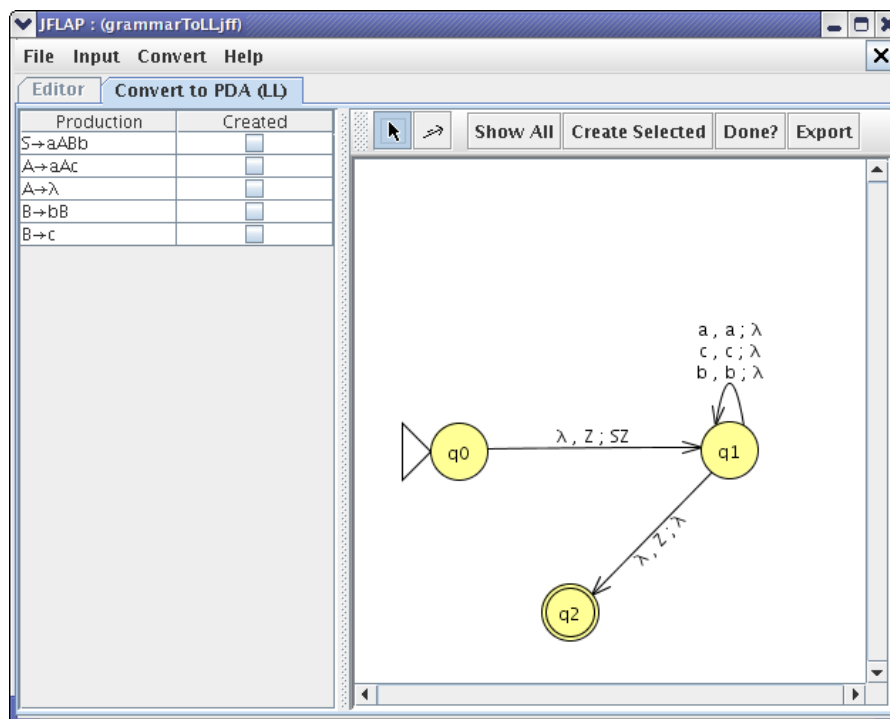
### Definition

We will convert a CFG to a PDA using the LL parsing method.

The idea behind the conversion from a CFG to an equivalent PDA (NPDA in this case) is to derive the productions through the stack. The conversion starts by pushing the start variable on the stack. Whenever there is a variable on the top of the stack, the conversion replaces the variable by its right side, thus applying a production toward deriving the string. Whenever a terminal is on top of the stack and the same symbol is the current input symbol, then the conversion pops the terminal off the stack and processes it as an input symbol. If the stack is emptied, then all the variables were replaced and all the terminals matched, so the string is accepted.

### How to Convert CFG to PDA (LL)

We will begin by loading the same grammar that we used for building the LL(1) parse table. You can view that grammar rule in **Build LL(1) Parse Table** tutorial. (or you can download the file, [grammarToLL.jff](#)). After loading the grammar, click on **Convert** and click **Convert CFG to PDA (LL)**. Now, your JFLAP window should look like this :

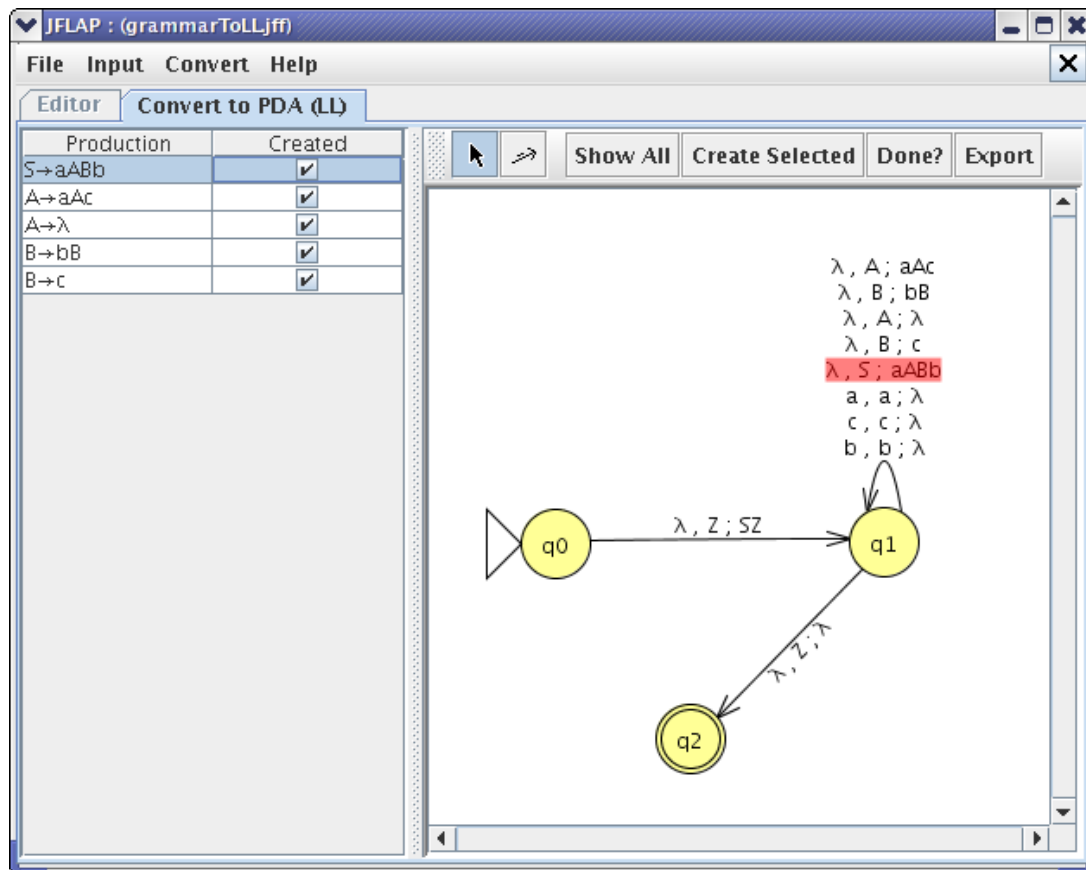


We need to add transitions between the states. Our goal is to reach state q2. We have already added the “popped off” loop transitions for matching terminals (the loops on state q1). We must

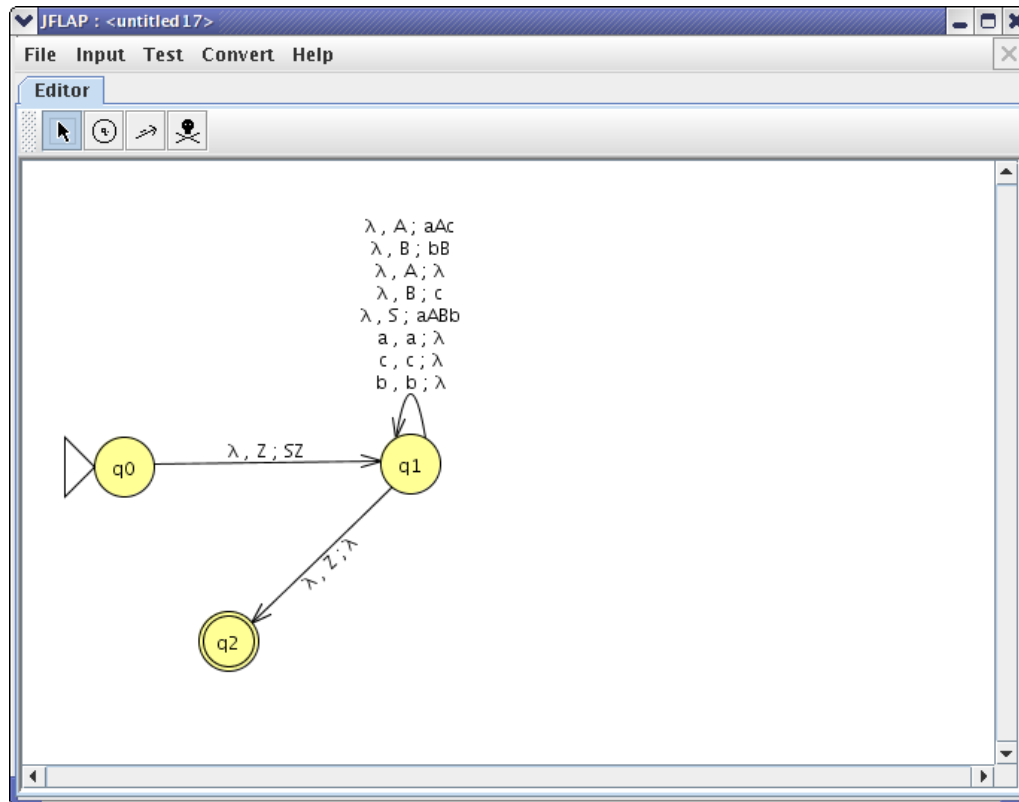
now add the “pushed on” loop transitions on state q1 for each production to get the terminals on the stack. For each production, the left side of the production is popped and the right side of the production is pushed. In this case, we can add the transition from q1 to q1 using “lambda, S : aABb”. This addition successfully portrays our production “S->aABb”.

**\*NOTE :** You can click on the production on the left side and click on **Create Selected** to add the corresponding transition in the PDA.

After entering all the transitions to the PDA, your JFLAP window should look similar to this :



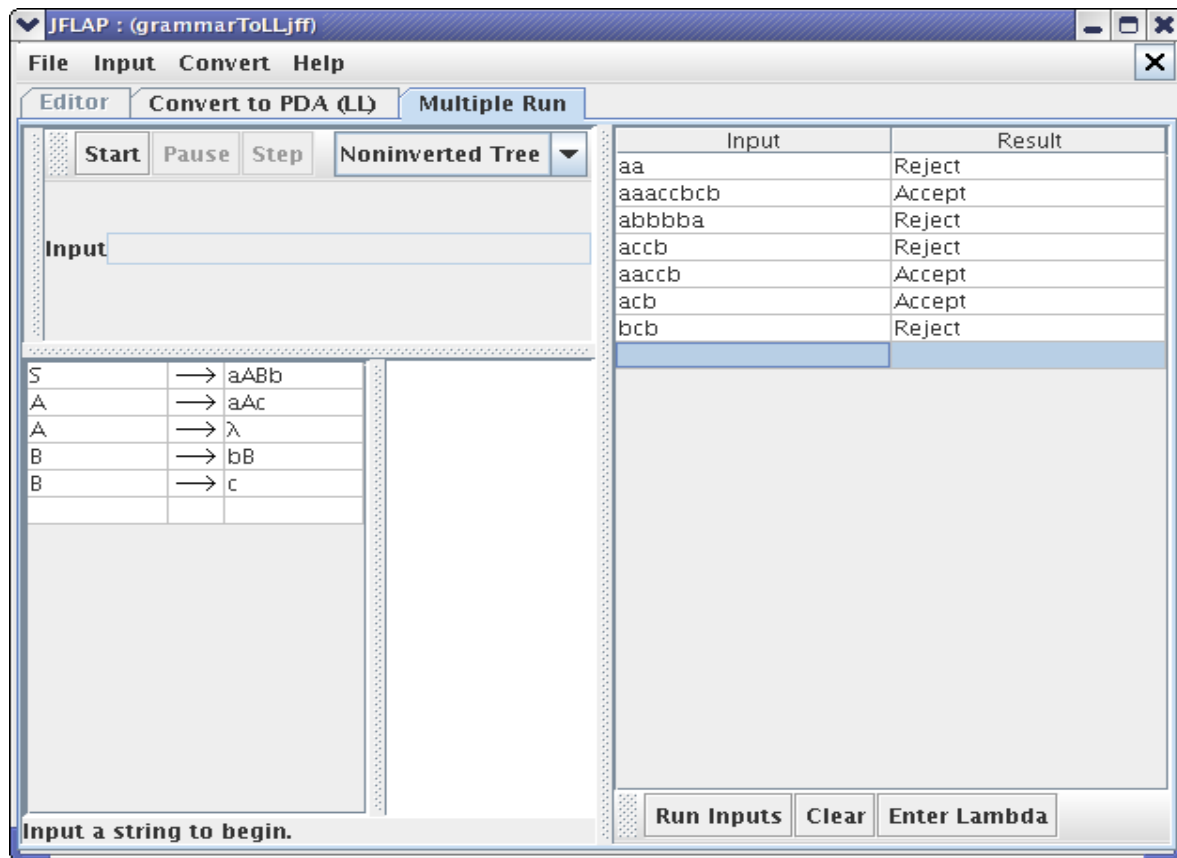
Now, we have successfully transformed the context-free grammar to a pushdown automaton. We can export this PDA and test it on some inputs to see if the converted PDA accepts the same string as our original CFG. After exporting the PDA, your new JFLAP window should look like :



Now let's run multiple run to test many strings. Click on **Input**, then click on **Multiple Run**. In the **Multiple Run** window, type input “aa”, “aaaccbcb”, “abbbba”, “accb”, “aaccb”, “acb”, “bcb”. In the Grammar window that we have opened for the conversion, click on **Input** then click on **Multiple Brute Force Parse** and enter the same strings into the input columns. After running the same inputs on PDA and Brute Force Parser, you should get the result :

Input	Result
aa	Reject
aaaccbcb	Accept
abbbba	Reject
accb	Reject
aaccb	Accept
acb	Accept
bcb	Reject

Run Inputs Clear Enter Lambda View Trace



Notice that both the PDA and our original grammar accepts/rejects the same strings.

## 16.Convert CFG to PDA (LR)

### Contents

#### Definition

#### How to Convert CFG to PDA (LR)

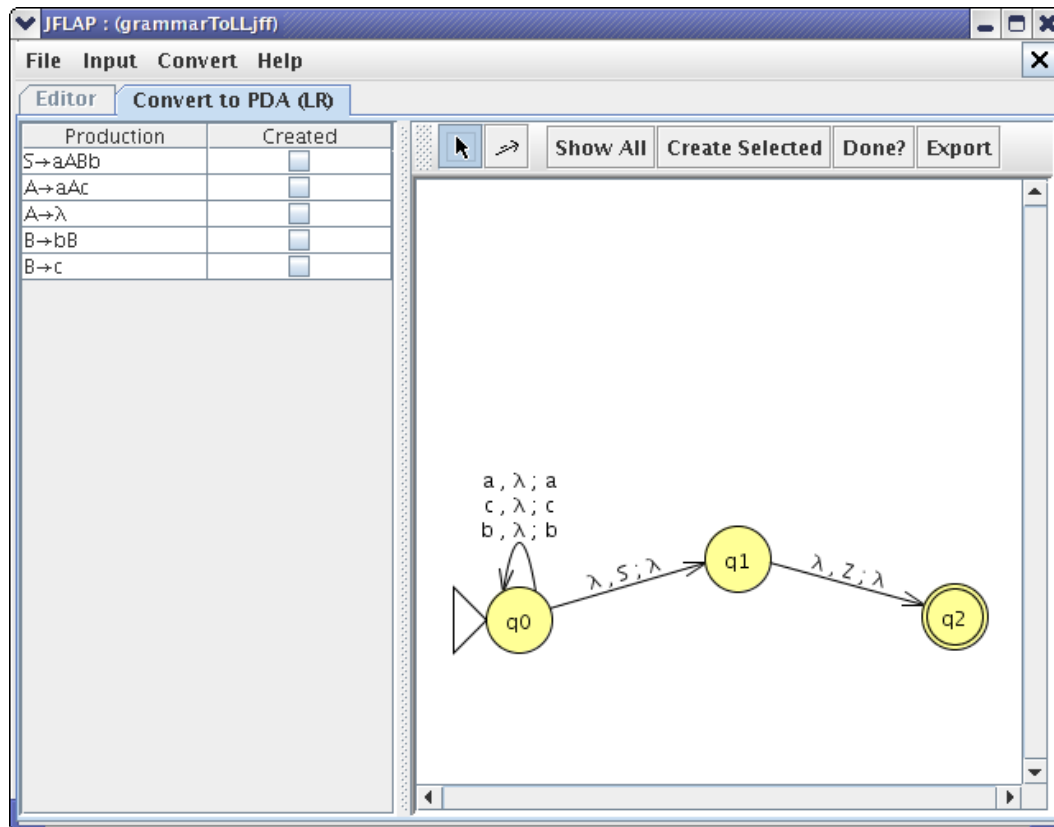
### Definition

We will convert a context free grammar into a pushdown automaton using the SLR(1) parsing method.

The idea behind the conversion from a CFG to an NPDA using the SLR(1) parsing method, is to push terminals from the strings on the stack, pop right-hand sides of productions off the stack, and push their left-hand sides on the stack, until the start variable is on the stack.

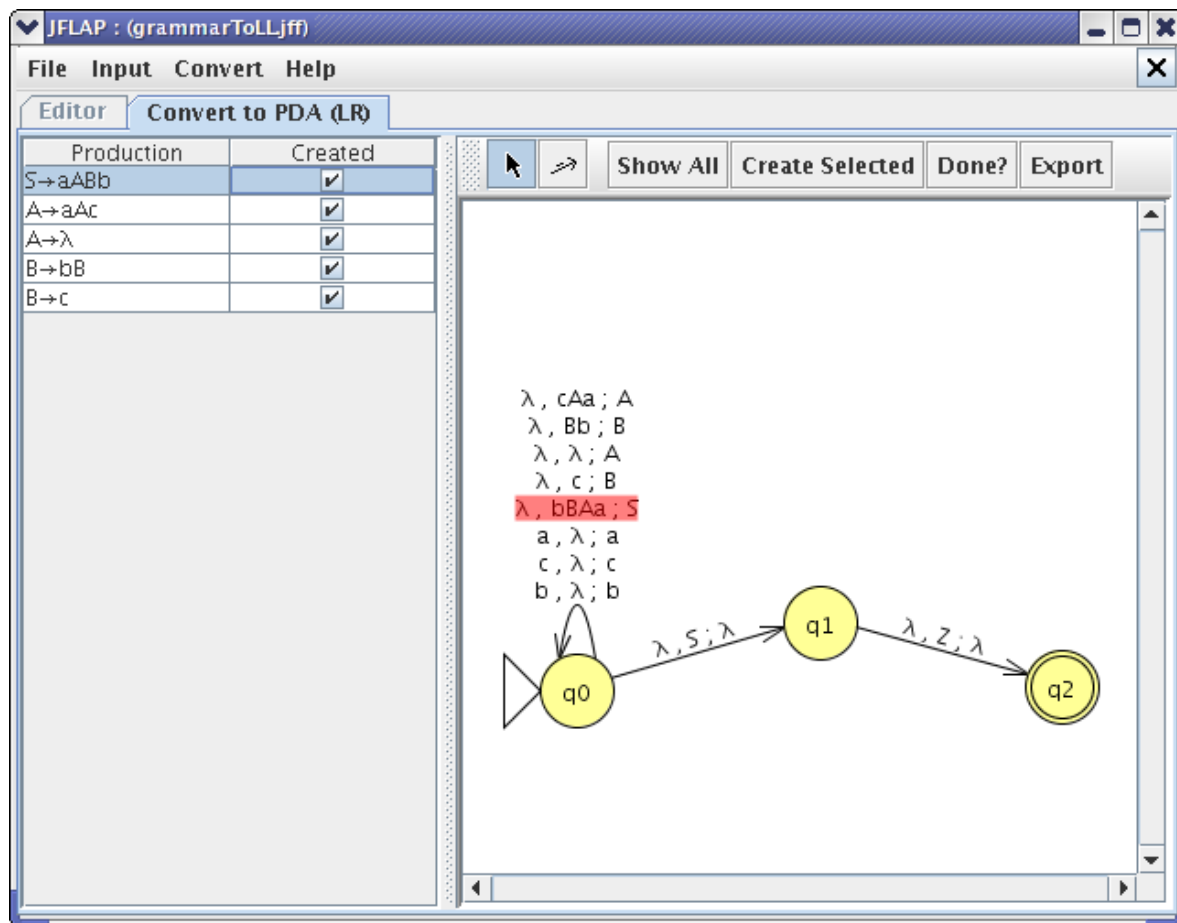
### How to Convert CFG to PDA (LR)

We will begin by loading the same grammar that we used for building the LL(1) parse table. You can view that grammar rule in our **Build LL(1) Parse Table** tutorial. (or you can download the file, [grammarToLL.jff](#)). After loading the grammar, click on **Convert** and click **Convert CFG to PDA (LR)**. Now, your JFLAP window should look like this :

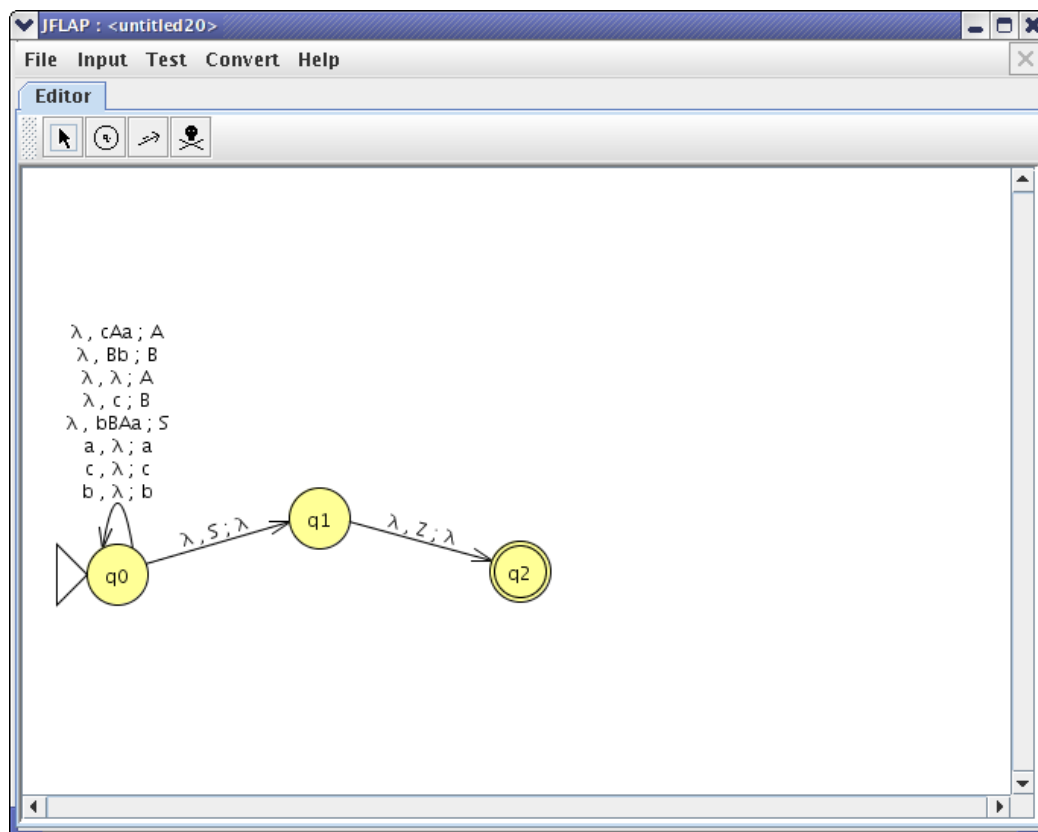


Just like the conversion to PDA using LL parsing, we need to add transitions between the states. However this time transitions are going to be quite different, because we are using a different parsing method. Still, our goal is to reach state  $q_2$ . For each terminal, add a loop transition on state  $q_0$  that reads the terminal in the input and pushes it on the stack. For each production, add a loop transition on state  $q_0$  that pops the right side of the production and pushes on the left side of the production from the stack. Therefore, for production “ $S \rightarrow aABb$ ”, we would put transition “ $\lambda, bBAa ; S$ ” on state  $q_0$ . We can convert all the other productions into transitions. After we have successfully completed them, the window will look like :

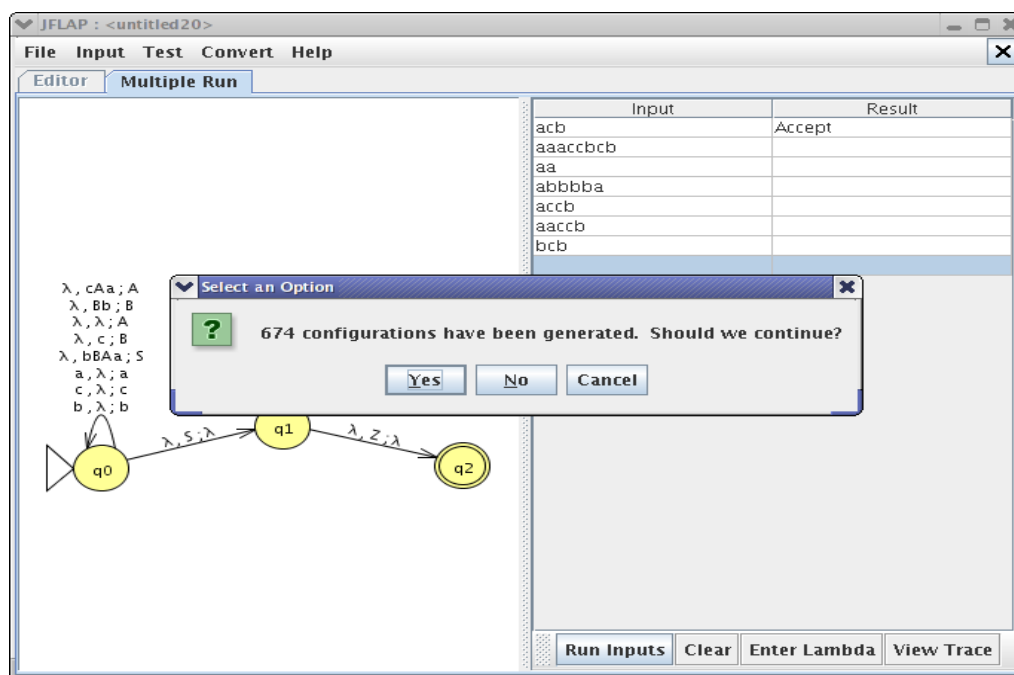
**\*NOTE :** You can click the production on the left side and click on **Create Selected** to add the corresponding transition in the PDA.



Now, we have successfully transformed the context-free grammar to a pushdown automaton. We can export this PDA and test it on some inputs and compare it to the original CFG. After exporting the PDA, your new JFLAP window should look like :



Now let's test many strings. Click on **Input**, then click on **Multiple Run**. In the **Multiple Run** window, type input “acb”, “aaaccbcb”, “aa”, “abbbba”, “accb”, “aaccb”, “bcb”. When you click **Run Inputs**, you might receive a message like this :

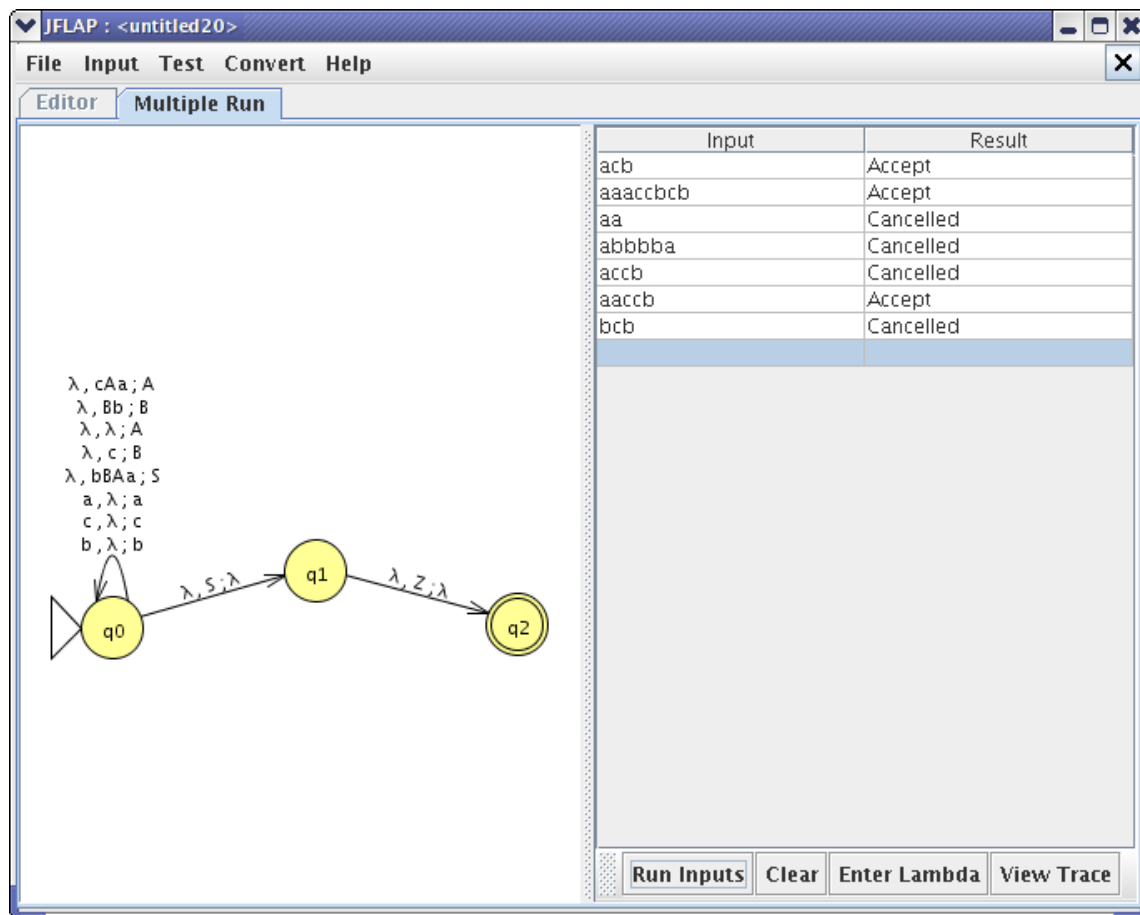


Notice that string “acb” is already accepted by PDA. However, when PDA is parsing the string “aaaccbcb”, it generated 674 configurations and still did not achieve the string yet. This does not

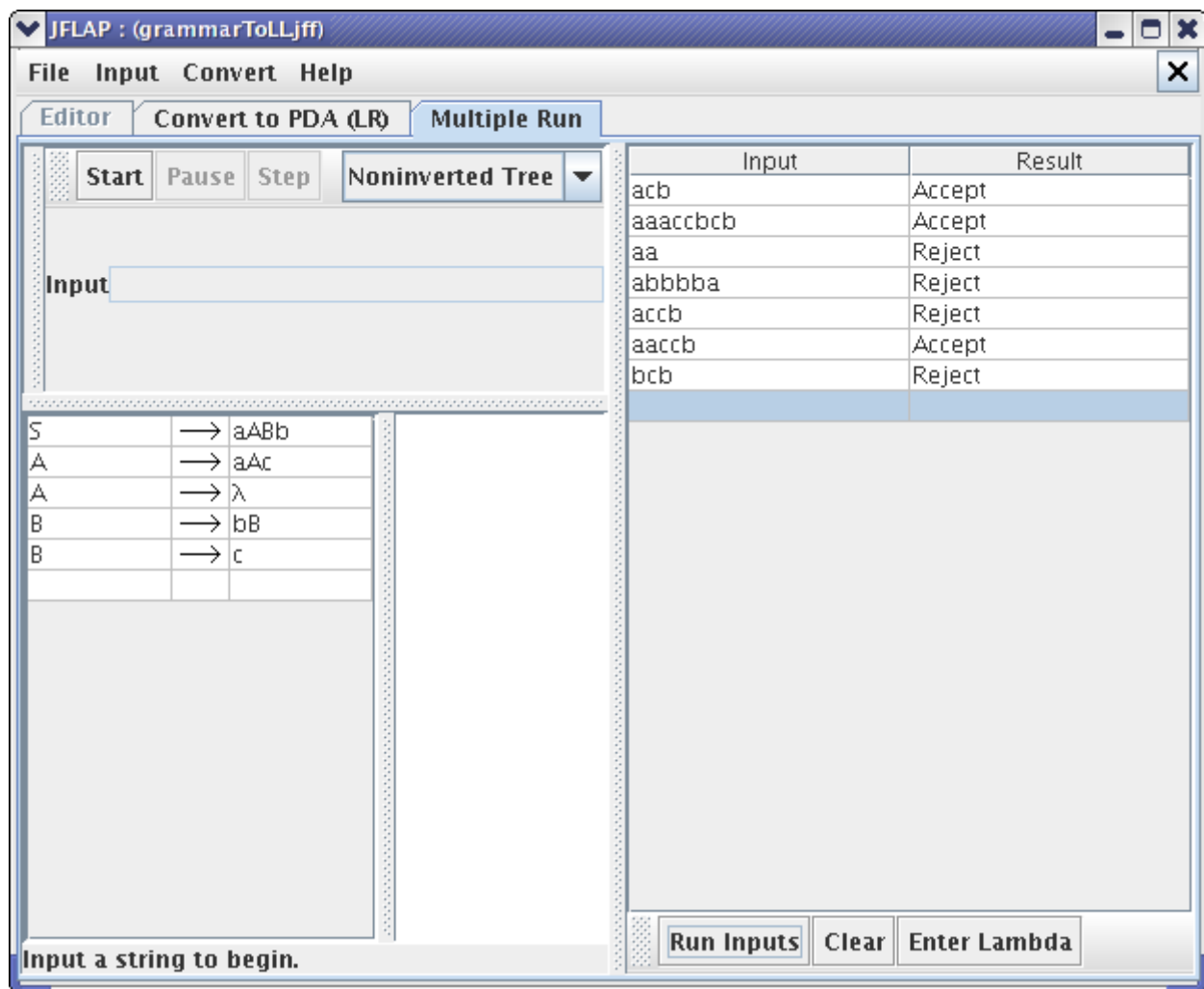


necessarily mean that the string is impossible to derive. But, it also implies that it could be the case that the string is impossible to derive. So let's press **Yes** button and see what happens. After clicking **Yes** to couple of times, you will notice that the string “aaaccbcb” is accepted eventually.

However, this is not always the case. Notice when you reach string “aa”, if you keep clicking on the **Yes** button, your configurations will get bigger and you will still not achieve the string. In this case, it would be wise not to continue. Select **No** for this string. After finishing the parsing, your window should look like :



If you run these same inputs on the grammar window using **Multiple Run**, you will get the same result. Your **Multiple Run** window should look like this after running :



## 17.Regular Expressions and Converting to a NFA

### Contents

#### Definition

#### Creating a Regular Expression

#### Converting to a NFA

### Definition

A regular expression is another representation of a regular language, and is defined over an alphabet (defined as  $\Sigma$ ). The simplest regular expressions are symbols from  $\lambda$ ,  $\emptyset$ , and symbols from  $\Sigma$ . Regular expressions can be built from these simple regular expressions with parenthesis, in addition to union, Kleene star and concatenation operators. In JFLAP, the concatenation symbol is implicit whenever two items are next to each other, and it is not explicitly stated. Thus, if one wishes to concatenate the strings “grass” and “hopper”, simply input “grasshopper”. The following is how JFLAP implements other special symbols for regular expressions:

( , ) are used to help define the order of operations

\* is the Kleene star

+ is the union operator

! is used to represent the empty string.

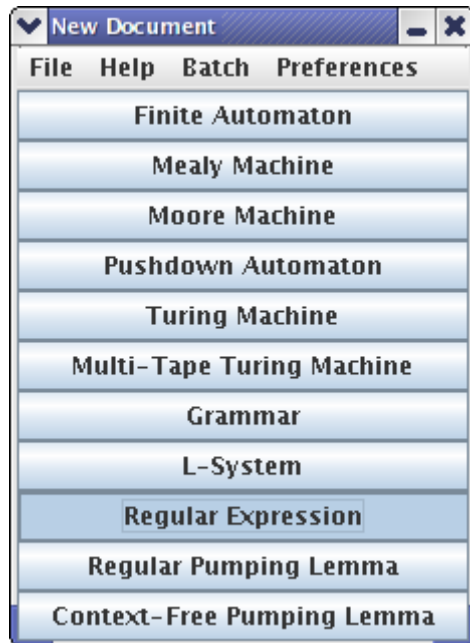
The following are a few examples of regular expressions and the languages generated using these operators:

- |                            |  |   |
|----------------------------|--|---|
| 1. $a+b+c = \{a, b, c\}$   | 4. $ab^* = \{a, ab, abb, abbb, \dots\}$                    | 7. $a+b^* = (a, \lambda, b, bb, bbb, \dots)$      |
| 2. $abc = \{abc\}$         | 5. $(ab)^* = (\lambda, ab, abab, ababab, \dots)$           | 8. $a+!^* = (a, \lambda)$                         |
| 3. $(!+a)bc = \{bc, abc\}$ | 6. $(a+b)^* = (\lambda, a, b, aa, ab, ba, bb, aaa, \dots)$ | 9. $(a+!)^* = (\lambda, a, aa, aaa, aaaa, \dots)$ |

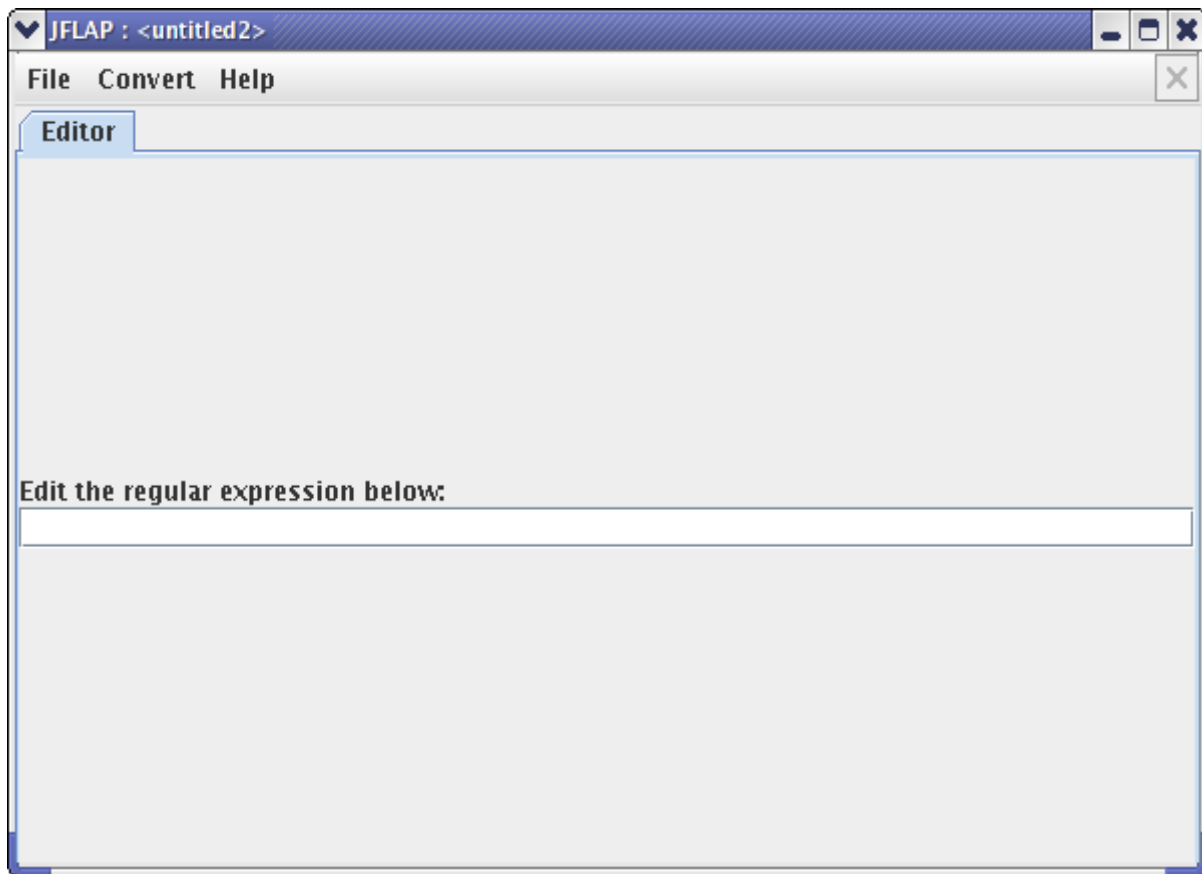
Since all regular languages accept finite acceptors, a regular expression must also be able to accept a finite automaton. There is a feature in JFLAP that allows the conversion of a regular expression to an NFA, which will be explained shortly. For knowledge of many of the general tools, menus, and windows used to create an automaton, one should first read the tutorial on [finite automata](#).

### Creating a Regular Expression

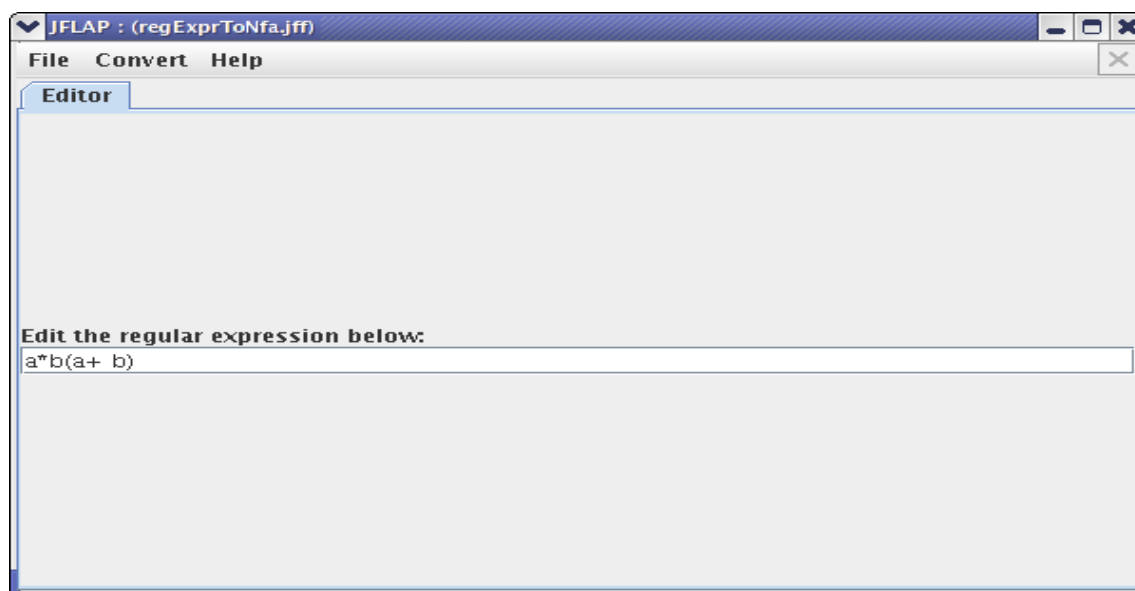
To create a new regular expression, start JFLAP and click the **Regular Expression** option from the menu, as shown below:



One should eventually see a blank screen that looks like the screen below.

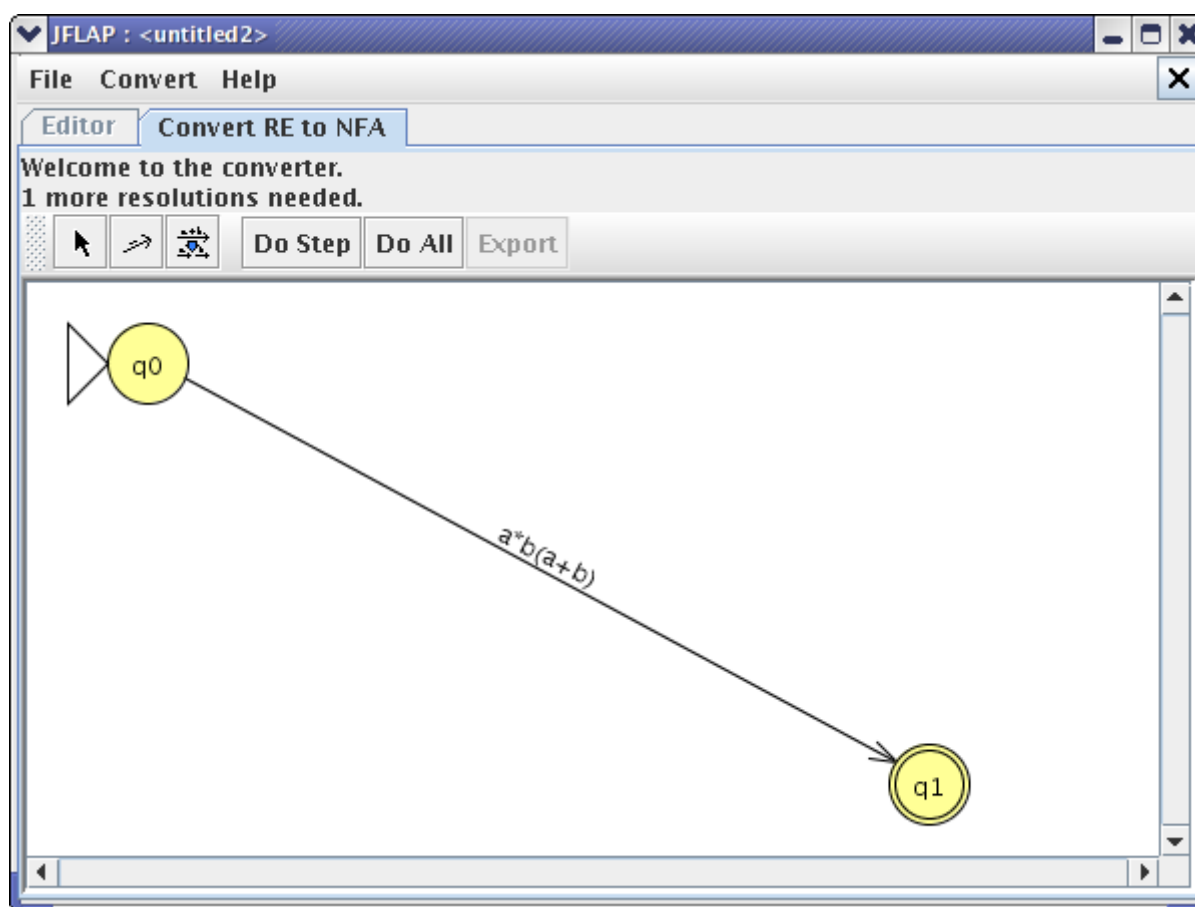


Now one can enter whatever expression that one wishes. Note that there should be no blanks in the regular expression, as any blank will be processed as a symbol of  $\Sigma$ . One such expression is in the screen below. One can either type it in directly, or load the file [regExprToNfa.jff](#).

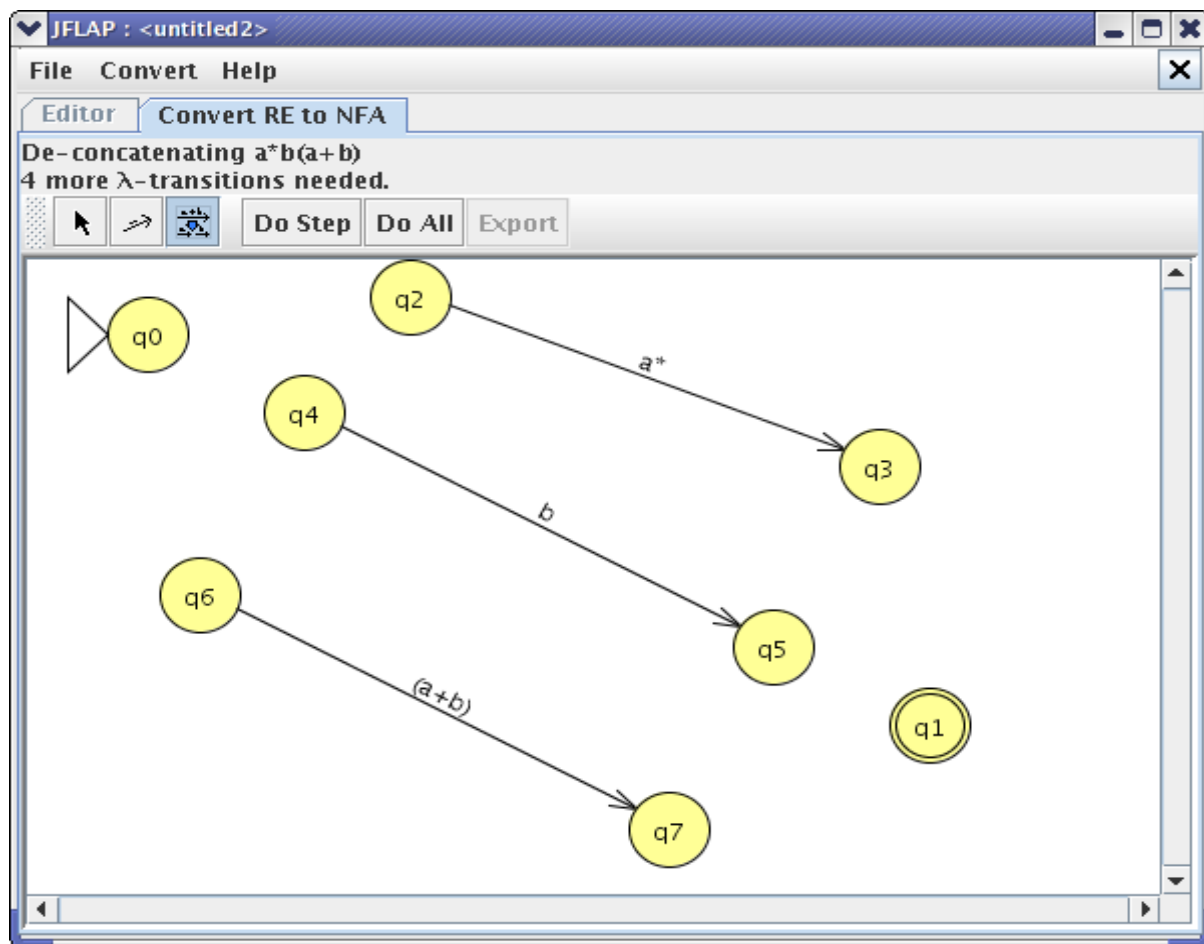


### Converting to a NFA

After typing in an expression, there is nothing else that can be done in this editor window besides converting it to an NFA, so let's proceed to that. Click on the "Convert → Convert to NFA" menu option. If one uses the example provided earlier, this screen should come up (after resizing the window a little).



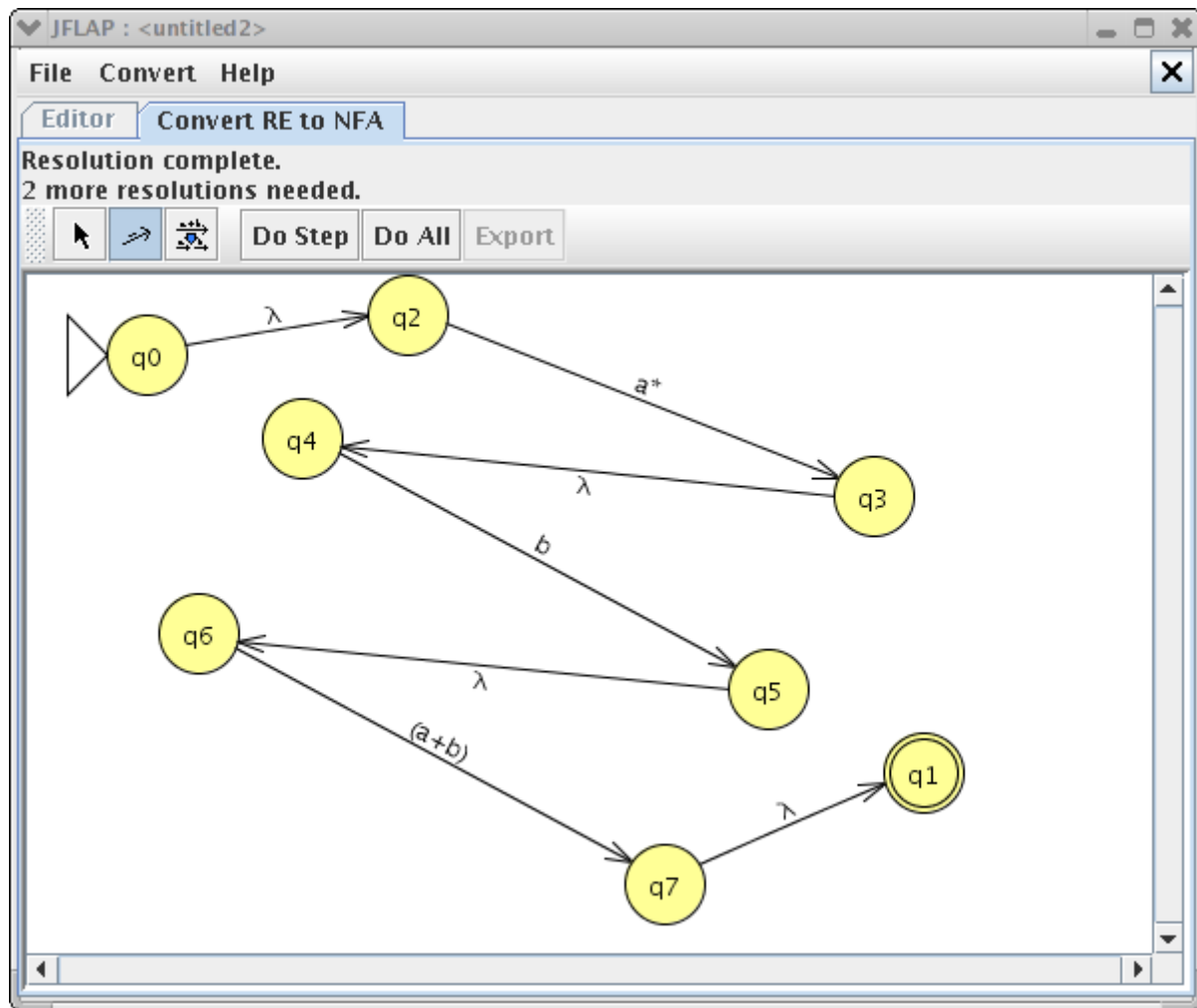
Now, click on the "(D)e-expressionify Transition" button (third from the left, to the immediate left of the "Do Step" button). Then, click on the transition from "q0" to "q1". You should now see, perhaps with a little resizing, a screen like this...



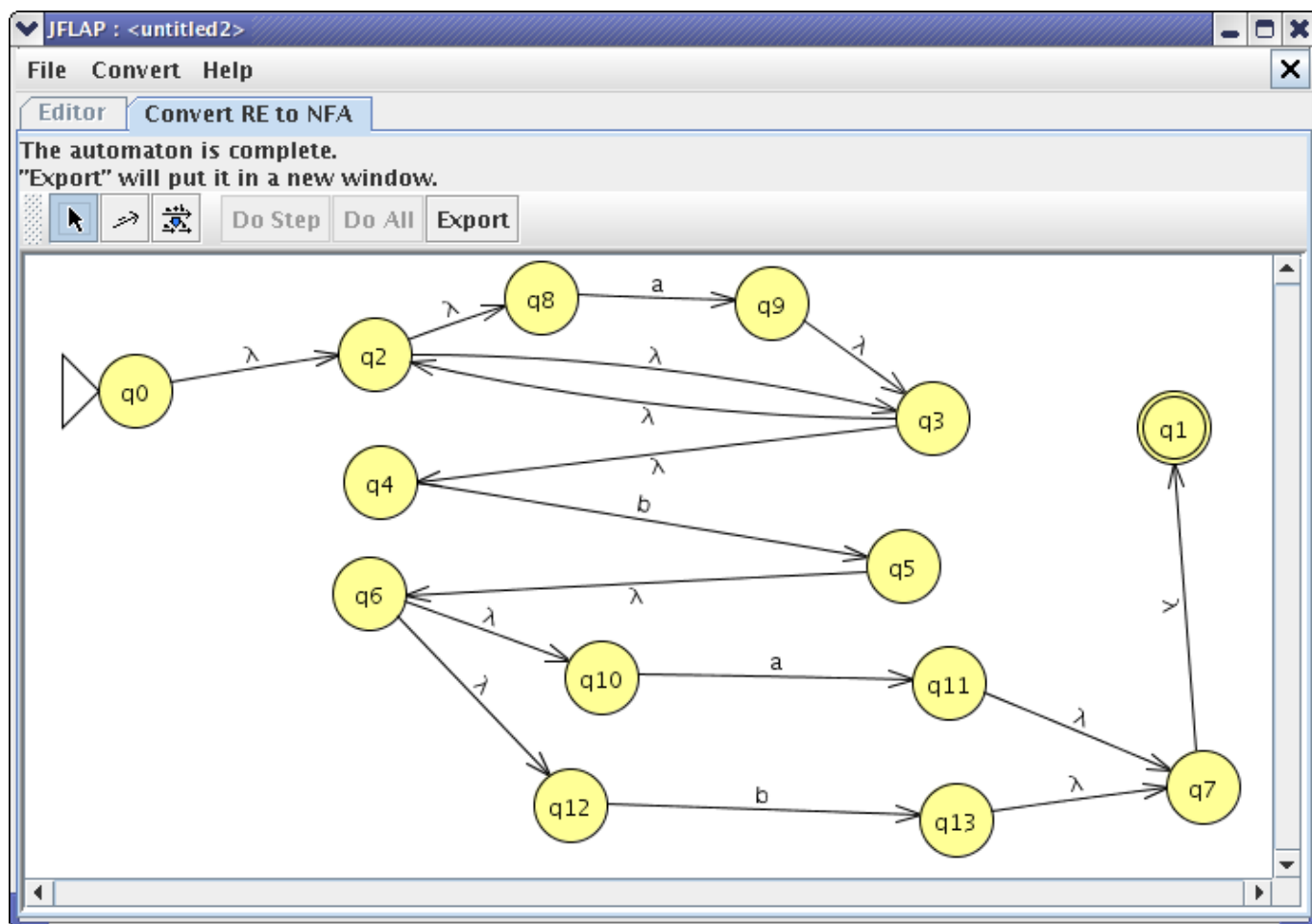
You're probably wondering what exactly you just did. Basically, you broke the regular expression into three sub-expressions, which were united into one expression by the implicit concatenation operator. Whenever you click on an expression or sub-expression through this button, it will subdivide according to the last unique operation in the order of operations. Thus, since concatenations are the last operation to be performed on the given expression, the expression is divided according to that operator.

Let's continue. Click on the second button from the left, the "(T)ransition Creator" button. Now, let's make our first transition. Create a transition from "q0" to "q2". You will not be prompted by a label, as in this mode only " $\lambda$ " transitions are created. These types of transitions are all that are needed to create a nondeterministic automaton. When finished, you should see a " $\lambda$ " transition between q0 and q2. Now, try to create a transition from q0 to q4. You will be notified that such a transition is "invalid". This is because, due to the concatenation operation between sub-expressions " $a^*$ " and " $b$ ", any " $b$ " must first process the " $a^*$ " part of the NFA. While it is possible to go to "q4" without processing any input, that will have to wait until the " $a^*$ " expression is decomposed. Thus, in order to get to "q4", we need to go through "q3", the final state of the " $a^*$ " expression. If you create a transition from "q3" to "q4", it will be accepted.

Since " $(a+b)$ " is the last expression, we will get to the final state after processing it. Because of this, try to establish a transition from "q7" to "q1". However, you should be warned that although the transition may be correct, you must create the transitions in the correct order. Thus, add the transition from "q5" to "q6" before adding the "q7" to "q1" transition. When done, your screen should resemble the one below.



Now, decompose the “a\*” expression. Two new states should be created, “q8” and “q9”, and you should add transitions from “q2” to “q8”, “q9” to “q3”, “q2” to “q3”, and “q3” to “q2”. You may wonder why we cannot just add a transition from “q0” to “q3”. While such a transition is legal, JFLAP forces the transition to go through “q2”, because that is the starting state for the “a\*” expression. Continue decomposing the expression using the tools provided. If you are stuck, feel free to use the “Do Step” button, which will perform the next step in the decomposition. If you wish to simply see the completed NFA, press the “Do All” button. You will probably have to move the states around in the screen and/or resize the screen so the decomposition looks good, whichever option you choose. In any event, when done, you should see a screen resembling the one below. Notice the indeterminate fork through “q6”, representing the union operator, where one can process either an “a” or a “b” transition. When finished, click the "Export" button, and you now have a nondeterministic finite automaton that you may use as you see fit.





## 18.Regular Pumping Lemmas

### Contents

Definition

Explaining the Game

Starting the Game

User Goes First

Computer Goes First

This game approach to the pumping lemma is based on the approach in Peter Linz's *An Introduction to Formal Languages and Automata*.

### Definition

JFLAP defines a regular pumping lemma to be the following. Let  $L$  be an infinite regular language. Then there exists some positive integer  $m$  such that any  $w$  that is a member of  $L$  with  $|w| \geq m$  can be decomposed as

$w = xyz$ ,

with

$|xy| \leq m$ ,

and

$|y| \geq 1$ ,

such that

$w_i = xy^i z$ ,

is also in  $L$  for all  $i = 0, 1, 2, \dots$

In other words, any sufficiently long string in  $L$  can be broken down into three parts such that any number of repetitions of the middle part (pumping the middle part) will still yield in a string in  $L$ .

### Explaining the Game

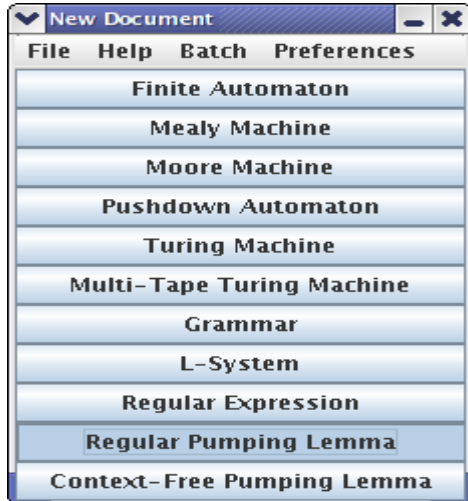
JFLAP treats the regular pumping lemma as a two-player game. In the game below, Player A is trying to find a  $xyz$  decomposition that is always in the language no matter what the  $i$  value is. Player B is trying to make it as hard as possible for player A to do so. If player B can pick a strategy such that he or she will always win regardless of player A's choices, meaning that no adequate decomposition exists, it is equivalent to proof that the language is not regular. The game is played like this:

- 1.Player A picks an integer for  $m$ .
- 2.Player B picks a string  $w$  such that  $w$  is a member of  $L$  and  $|w| \geq m$ .
- 3.Player A picks the partition of  $w$  into  $xyz$  such that  $|xy| \leq m$  and  $|y| \geq 1$ .
- 4.Player B picks an integer  $i$  such that  $xy^i z$  is not a member of  $L$ . If player B can do so player B wins, otherwise, player A wins.

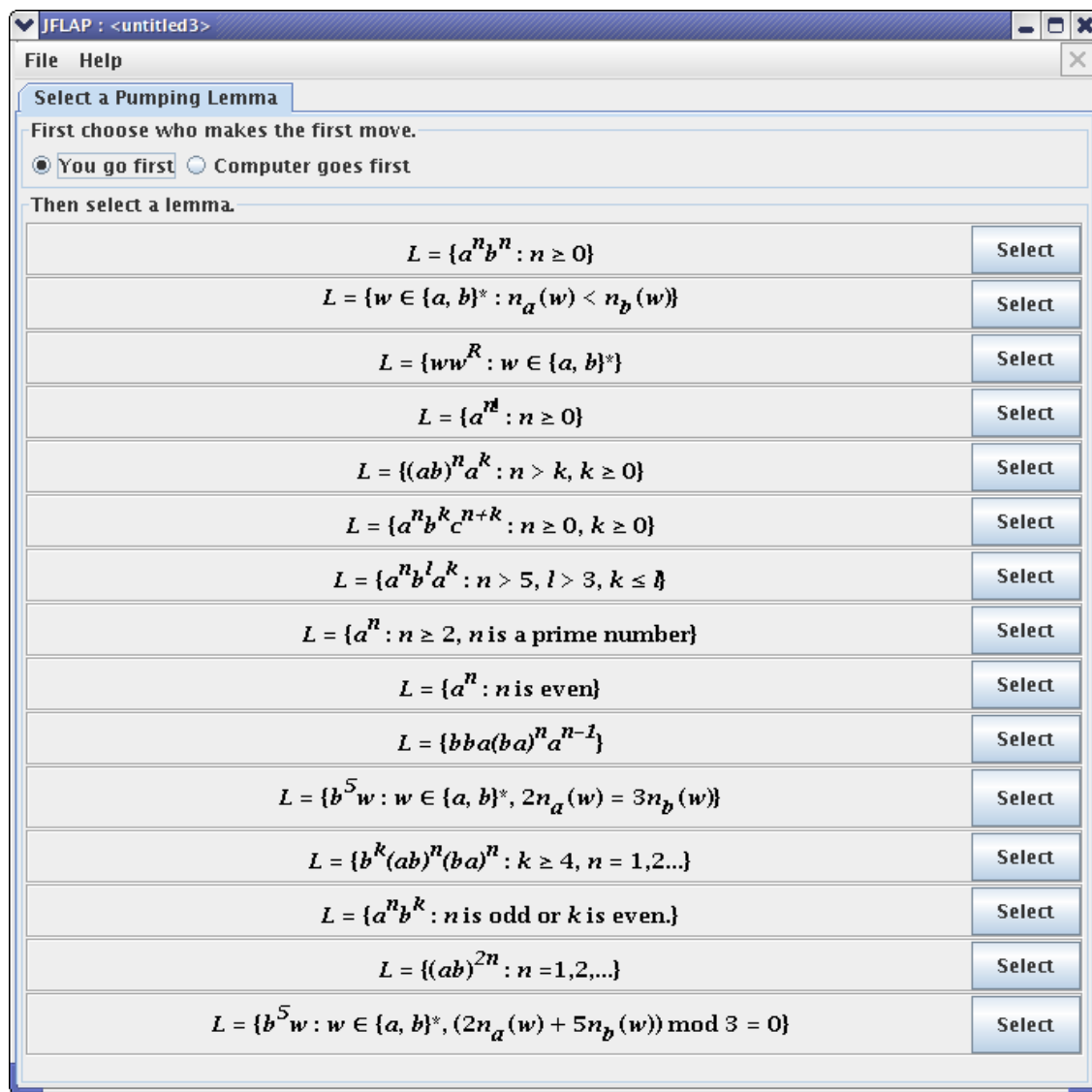
There are two possible modes for the game, when the user goes first and when the computer goes first. In the first mode, the user is player A and the computer is player B, and thus the user should be trying to find an acceptable decomposition to pump. In the second mode, the user is player B and the computer is player A, and thus the user should be trying to prevent the computer from generating a valid decomposition.

### Starting the Game

To start a regular pumping lemma game, select **Regular Pumping Lemma** from the main menu:

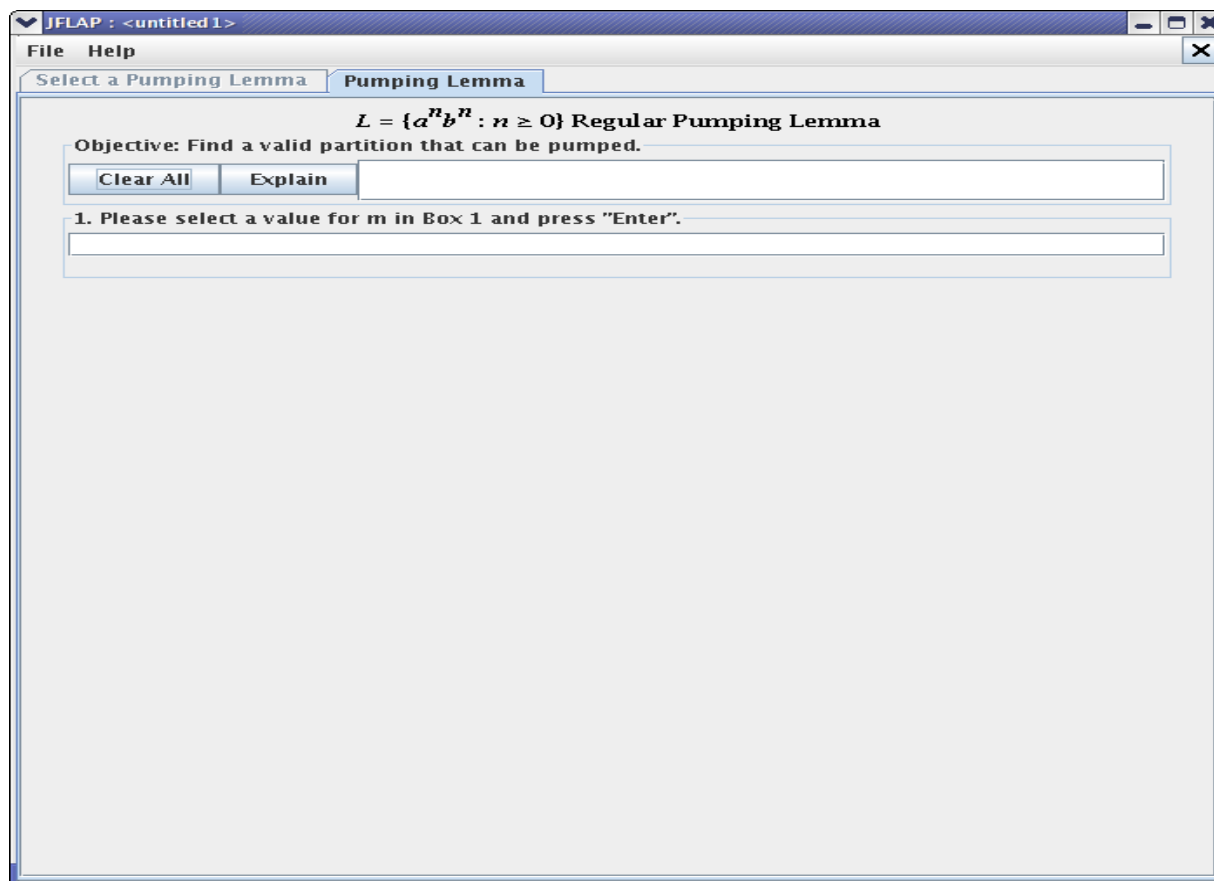


You will then see a new window that prompts you both for which mode you wish to utilize and which language you wish to work on. The default mode is for the user to go first. A list of languages is also shown, some of which are regular and some of which are not regular. To proceed to a language, click on the appropriate "Select" button, and a new screen will come up based on the language chosen and the mode selected when the button was pushed.



### User Goes First

Let's do an example for when the user goes first. Since the “You go first” option is already selected, we do not need to do anything in the top panel. Now we need to choose a language. Let's try the first language on the screen,  $L = \{a^n b^n : n \geq 0\}$ . Go ahead and press the “Select” button to the right of the language. The following screen should come up.



You may be curious about all the various items on the screen. We will touch all of them in time, but for now, let's play the game. We see that our objective in this mode is to find a valid partition, and that the first step that we need to preform is to choose a value for  $m$ . Go ahead and enter a fairly large value for  $m$ , such as 20, in the box where you are prompted. If the value is too large, the following message should appear in the panel where you entered  $m$ .

1. Please select a value for  $m$  in Box 1 and press "Enter".

20

Please enter a positive integer in range [4, 18] for best results.

JFLAP will only accept values for  $m$  in a finite range, both so  $m$  is large enough and so unnecessarily large values are avoided. Now, go ahead and enter "4", which is a value of  $m$  in accepted range. The error message should disappear, and the following two panels should now be visible on the screen under the  $m$  panel.

2. I have selected  $w$  such that  $|w| \geq m$ . It is displayed in Box 2.

aaaabbbb

3. Select decomposition of  $w$  into  $xyz$ .

x: |x|: 0

y: |y|: 0

z: |z|:

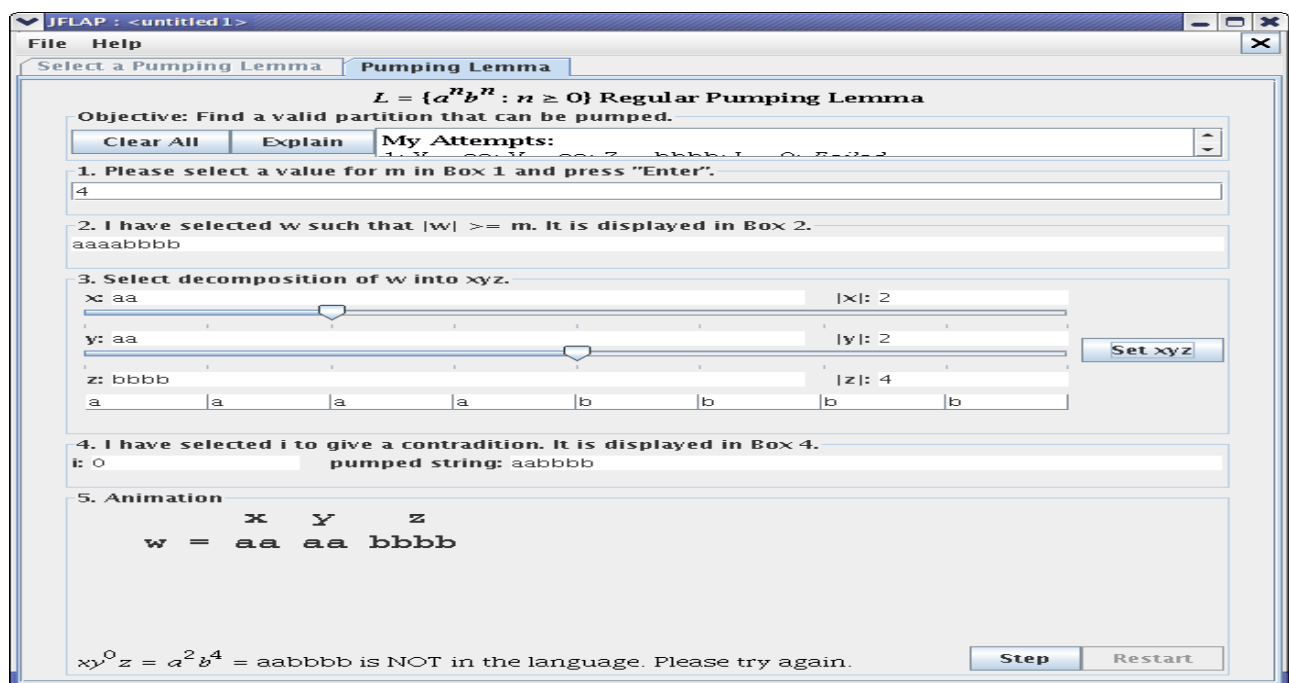
a a a a b b b b

Condition violated:  $|y| > 0$

Set xyz

The computer has chosen a value for  $w$ , and now the game prompts you to decompose the string. You can adjust the  $x$ ,  $y$ , and  $z$  substrings by using the sliders to set the boundaries between them. The first slider will set the boundary between  $x$  and  $y$ , and the second will set the boundary between  $y$  and  $z$ . If the decomposition is acceptable, the “Set xyz” button will become visible and the message at the bottom of the panel will inform you that you can set the button. If it is unacceptable, the message at the bottom will inform you what the problem is with the current decomposition.

Let's have  $x$  and  $y$  both equal “aa”. Go ahead and slide the top slider over two spaces. You should notice that the bottom slider also moves too. Then, slide the bottom slider over two more spaces, and then click the visible “Set xyz” button. You should notice that the decomposition for each substring appears in the appropriate boxes as you move the sliders, in addition to the lengths of the substrings (note however that all fields may not be filled in if the decomposition is invalid). When finished, the message in the decomposition panel will disappear, the last two panels will become visible, and the whole screen will look like this.



There are a number of things that you should notice. First, in the panel below the decomposition panel, we see that the computer has chosen a value for  $i$ , which is 0. It also shows the pumped string, which is “aabbbb”. The panel below informs you that you have lost, as the pumped string is not in the language. This panel also allows you to see an animation of how the pumped string is assembled. Press the “Step” button to see each step in the animation, and the button will lose its visibility when all steps have been completed. “Restart”, if visible, will restart the animation at the beginning. When the animation is finished, the bottom panel should look like this.

### 5. Animation

$x \quad y \quad z$   
 $w = aa \ aa \ bbbb$   
 $aabbbb$

Unfortunately,  $xy^0z = a^2b^4 = aabbbb$  is NOT in the language. Please try again.

Step

Restart

You should also notice that there is new information in the text box in the top panel. What is listed here is all the attempts that you have made to win this game. Scroll down a little in this text box and you should see your decomposition,  $i$  value, and result (either *Failed* or *Won*) for each attempt.

#### My Attempts.

1:  $X = aa; Y = aa; Z = bbbb; i = 0; Failed$

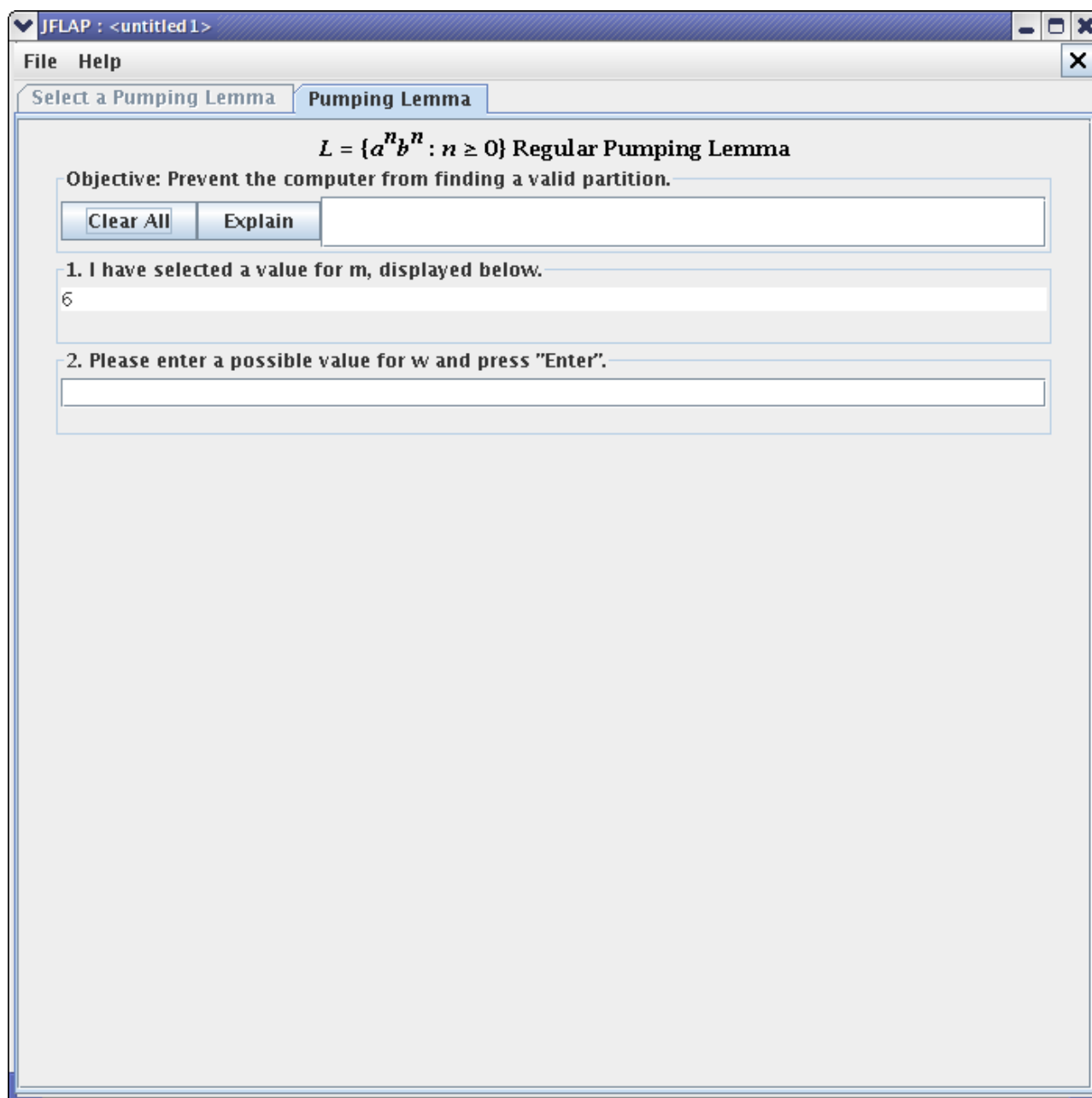
Since we have made only one attempt so far, only one attempt is listed. If we try a few more times, more attempts will be added to the list. The more recent the attempt, the closer it will be to the top of the list. There are a variety of ways to try again. If we are happy with our  $m$  and  $w$  values, we can simply choose a new decomposition in the decomposition panel and press the “Set xyz” button. If, however, we wish to change the  $m$  value, we can either enter a new value in the  $m$  panel directly or press the “Clear All” button, which is in the top panel, and then enter a new  $m$  value. If you press the “Clear All” button, notice that the list of attempts remains visible. After entering a new  $m$  value and after the computer picks a new  $w$  value, you would then choose a new decomposition. The visibility of all panels will be updated to reflect the current stage of the game.

After a few attempts, you may notice that you never win, no matter what  $m$  value you choose or which decomposition you pick. However, if you don't spot the strategy that the computer is employing, you may wonder whether you are simply choosing your values poorly or whether it is indeed impossible to win. Once you have made enough attempts, feel free to press the “Explain” button in the top panel. You will now see, in the text box where the attempts are listed (you can still see them if you scroll down further), whether a valid partition exists. Below this information is an informal, intuitive proof about why there is or is not a valid partition. For this example, there is indeed no valid partition. To get rid of the proof and information about the possibility of a valid partition, simply click on the “Clear All” button.

The example you just generated is available in [regUserFirst.jff](#).

### Computer Goes First

When finished, dismiss the tab, and you will return to the language selection screen. In the top panel, click on “Computer goes first.” Then, choose the first language,  $L = \{a^n b^n : n \geq 0\}$ , again. The following screen should come up.



You will notice a few things different from when you went first. First, the objective has changed. Your purpose is to prevent the computer from finding a valid partition. Second, the messages for each step in the game have changed, reflecting the different tasks you must perform as player B. The computer has chosen a value for  $m$  within the permissible range, in this case 9. You are prompted for a value for  $w$ . Go ahead and enter the string “aaaaabbbbb”, which is the smallest string in the language where  $|w| \geq m$ . (Tip: if you get a large value for  $m$ , such as 18, and you wish for a smaller one, press “Clear All” repeatedly until one is generated.) If for some reason you enter a string that is either not in the language or where  $|w| < m$ , the appropriate error message will appear in the  $w$  panel. When finished, press enter, and the following two panels will become visible.

3. I have decomposed  $w$  into the following...

$X = aa; Y = aaabbbb; Z = b$

---

4. Please enter a possible value for  $i$  and press "Enter".

$i$ :  pumped string:

The computer has chosen a decomposition based on the  $w$  value that you entered. Now, given this decomposition, it is your duty to choose an  $i$  value. Go ahead and enter 13 for  $i$ . However, instead of progressing to the next panel, the following error message should appear.

4. Please enter a possible value for  $i$  and press "Enter".

$i$ :  pumped string:

Please enter a positive integer in range [0, 2...12] for best results.

For all languages JFLAP supports, you can only enter either 0 or a value between 2 and 12 for  $i$ . A value of 1 will result in a pumped string equal to  $w$ , and thus is always in the language, so it is excluded.  $i$  values larger than 12 are excluded so the pumped string doesn't get too large. Thus, go ahead and enter 2 for  $i$ , which is in the permissible range. When finished, the screen should resemble the one below. This example is also available in [regCompFirst.jff](#).

JFLAP : <untitled 1>

File Help

Select a Pumping Lemma Pumping Lemma

$L = \{a^n b^n : n \geq 0\}$  Regular Pumping Lemma

Objective: Prevent the computer from finding a valid partition.

Clear All Explain My Attempts:

1. I have selected a value for  $m$ , displayed below.

9

2. Please enter a possible value for  $w$  and press "Enter".

aaaaabbbbb

3. I have decomposed  $w$  into the following...

$X = aa; Y = aaabbbb; Z = b$

4. Please enter a possible value for  $i$  and press "Enter".

$i$ :  pumped string: aaaaabbbbbaabbbbb

5. Animation

	$x$	$y$	$z$
$w =$	aa	aaabbbb	b

$xy^2z = a^5 b^4 a^3 b^5 = aaaaabbbbbaabbbbb$  is NOT in the language. YOU WIN!

Step Restart

You've won, because the computer could not choose a valid decomposition. In fact, as seen through the proof earlier, it never will find one with this language, so you will always win. Go ahead and make a few more attempts to familiarize yourself with the format, however. You can enter a new  $w$  value if you want to keep the same  $m$  value, or just press enter in the  $w$  text box if you want to keep both values. You can enter a new  $i$  value if you want to retain  $m$ ,  $w$ , and the decomposition. Panel visibility will adjust itself accordingly.



## 19.Context-Free Pumping Lemmas

### Contents

Definition

Explaining the Game

Starting the Game

User Goes First

Computer Goes First

This game approach to the pumping lemma is based on the approach in Peter Linz's *An Introduction to Formal Languages and Automata*.

Before continuing, it is recommended that if you read the tutorial for regular pumping lemmas if you haven't already done so. It explains many of the basics of the game, both when the user goes first and when the computer goes first. This tutorial will primarily focus on those features which differentiate between context-free and regular pumping lemmas.

### Definition

JFLAP defines a context-free pumping lemma to be the following. Let  $L$  be an infinite context-free language. Then there exists some positive integer  $m$  such that any  $w$  that is a member of  $L$  with  $|w| \geq m$  can be decomposed as

$$w = uvxyz,$$

with

$$|vxy| \leq m,$$

and

$$|vy| \geq 1,$$

such that

$$w_i = uv^i xy^i z,$$

is also in  $L$  for all  $i = 0, 1, 2, \dots$

### Explaining the Game

JFLAP treats the context-free pumping lemma in a two-player game. In the game below, Player A is trying to find a  $uvxyz$  decomposition that is always in the language no matter what the  $i$  value is. If player B can pick a strategy such that he or she will always win regardless of player A's choices, meaning that no adequate decomposition exists, it is equivalent to proof that the language is not context-free. The game is played like this:

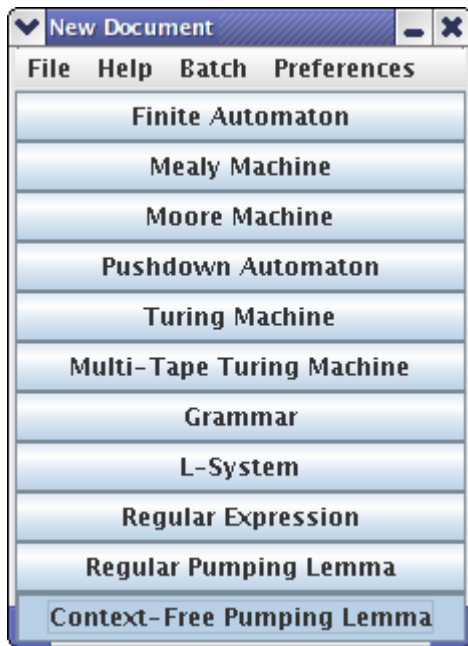
- 1.Player A picks an integer for  $m$ .
- 2.Player B picks a string  $w$  such that  $w$  is a member of  $L$  and  $|w| \geq m$ .
- 3.Player A picks the partition of  $w$  into  $uvxyz$  such that  $|vxy| \geq m$  and  $|vy| \geq 1$ .

4. Player B picks an integer  $i$  such that  $uv^i xy^i z$  is not a member of  $L$ . If player B can do so player B wins, otherwise, player A wins.

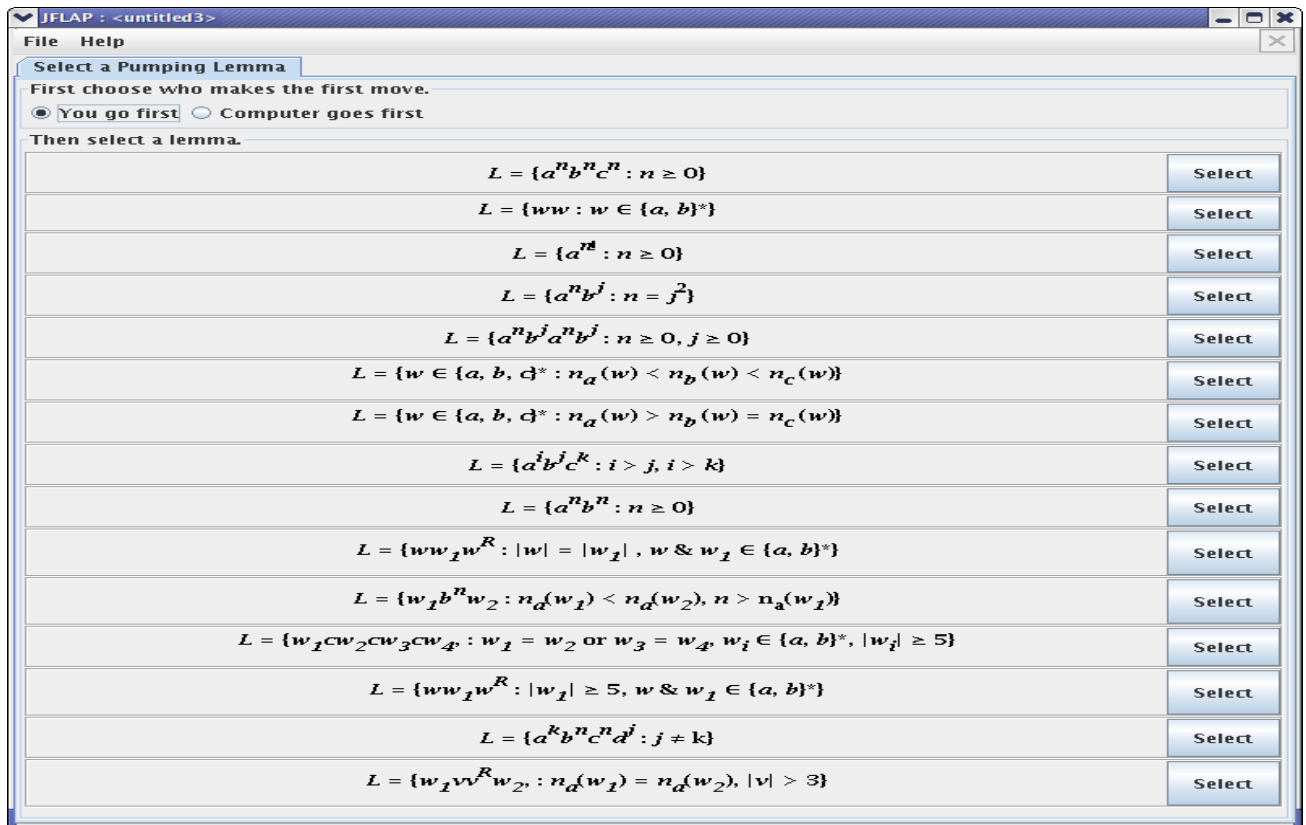
There are two possible modes for the game, when the user goes first and when the computer goes first. In the first mode, the user is player A and the computer is player B, and thus the user should be trying to find an acceptable decomposition to pump. In the second mode, the user is player B and the computer is player A, and thus the user should be trying to prevent the computer from generating a valid decomposition.

### Starting the Game

To start a context-free pumping lemma game, select **Context-Free Pumping Lemma** from the main menu:

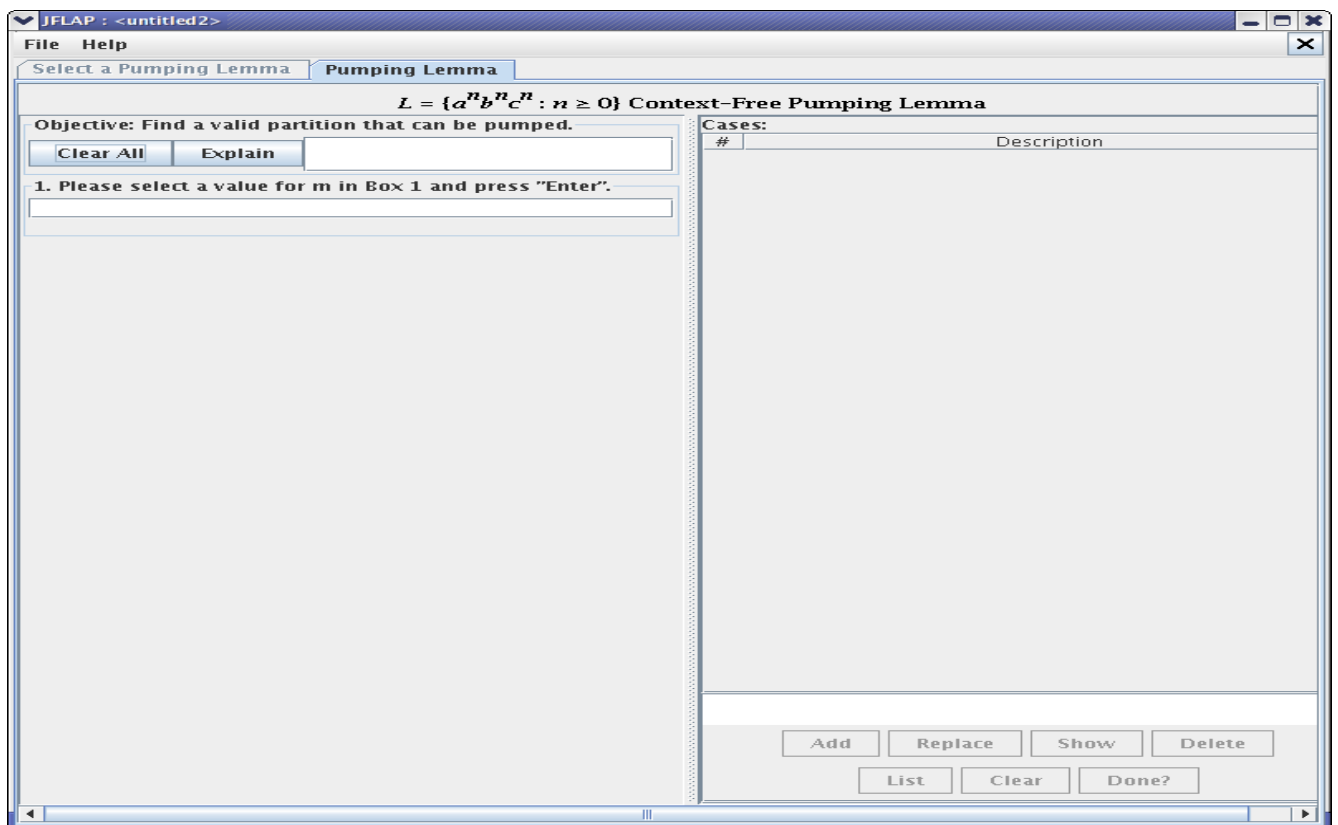


The following screen should come up. It has the same functionality as the corresponding screen for regular pumping lemmas, except this time it includes some languages which are context-free and some that are not.

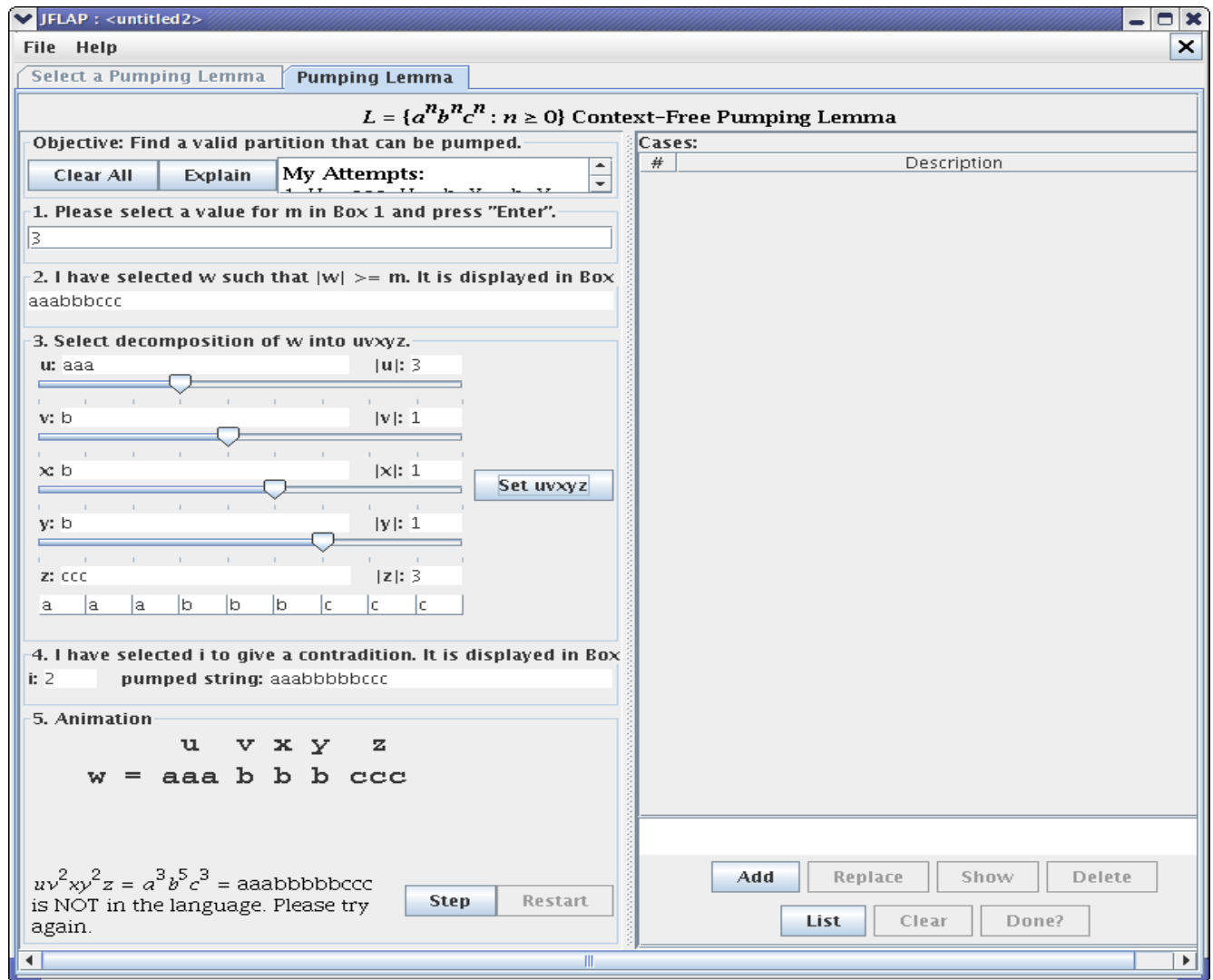


## User Goes First

Let's do an example for when the user goes first. Since the "You go first" option is already selected, we do not need to do anything in the top panel. Now, let's choose the first language in the list,  $L = \{a^n b^n c^n : n \geq 0\}$ . Go ahead and press the "Select" button to the right of the language. The following screen should come up.



You may notice that the left panel resembles the screen that we used for regular pumping lemmas. You may also wonder about the right panel, which is the case panel. Not all the languages listed use this panel, and the left panel takes up the whole screen for those that do not. However, this one does utilize it, and we will interact with it in time. First, however, play the game that is identical to the one played with regular pumping lemmas, except that instead of composing into  $xyz$  you will decompose into  $uvxyz$ . Use a value of 3 for  $m$  and a decomposition of  $u = \text{"aa"}, v = \text{"a"}, x = \text{"b"}, y = \text{"b"}, \text{ and } z = \text{"bccc"}$ . When finished, your screen should resemble the one below.



We failed to find an adequate decomposition this time. However, we have checked one possible case for our decomposition, which is when  $v$  has at least one "a" and "b" and when  $y$  has a "b". In many of the context-free languages supported by JFLAP (although not all of them), the languages support the storage and implementation of cases. This language is one of them. Thus, we can add the case that we have chosen to the case panel on the right. Click on the "Add" button, and a new line representing this case should become visible in the right panel.

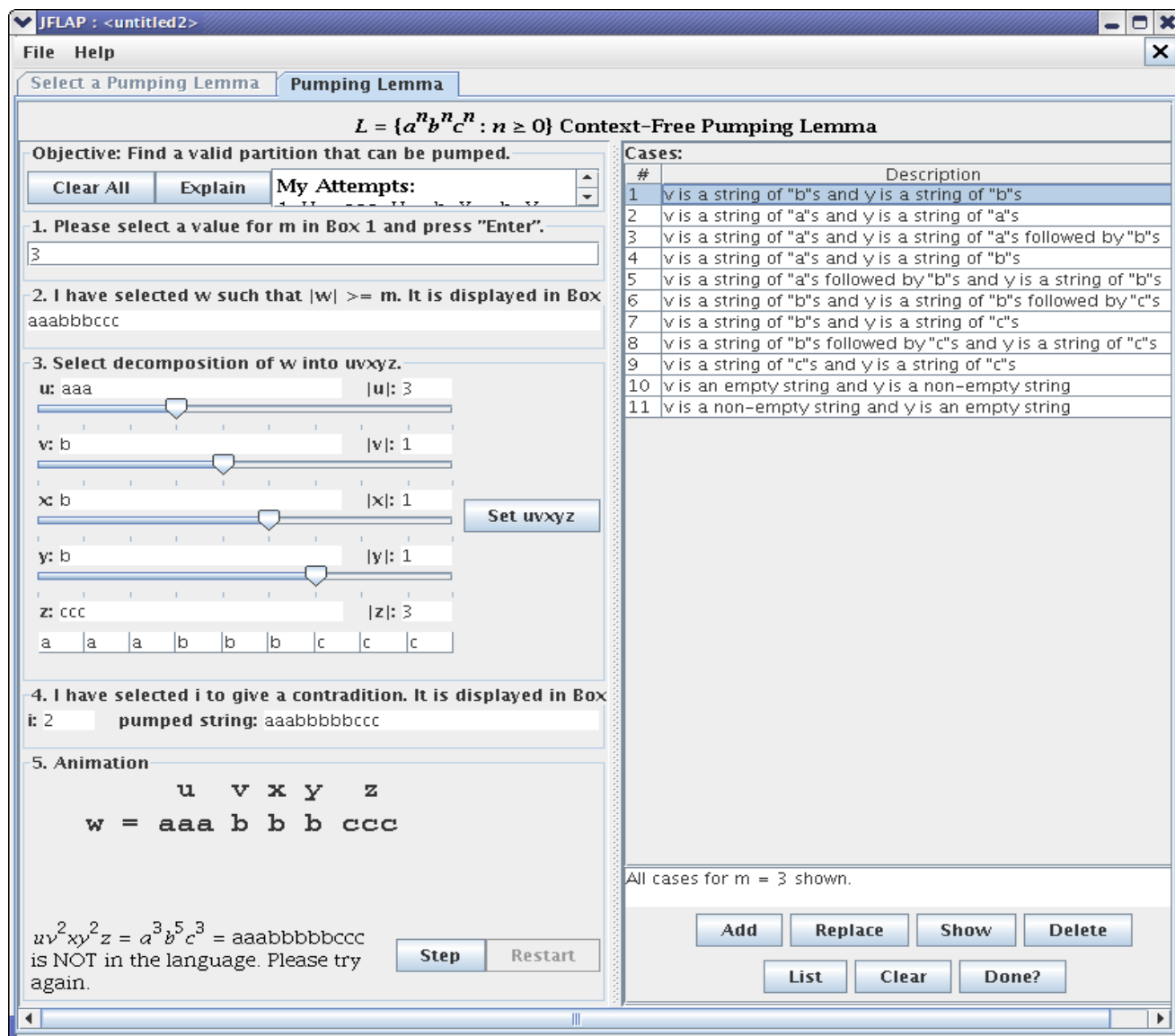
Cases:	
#	Description
1	v is a string of "a"s and y is a string of "b"s

You should also notice that all the buttons at the bottom of the case panel are now visible. What each button does is as follows:

- “Add” will add the current attempt to the case panel if the attempt represents a new case. If the case is already listed in the case panel, you are informed with a message in the text box above the buttons that that case resembles an existing case in the panel (telling you the number).
- “Replace” will replace the attempt stored in the highlighted case with the current attempt. Since there is only one case at present, it is already highlighted, but if there were more, you can highlight them by clicking on the relevant case with the mouse. If the current attempt is an invalid example of a particular case, you will be informed of that fact.
- “Show” will show on the left panel the current attempt stored in the highlighted case.
- “Delete” will remove the highlighted case from the case panel.
- “Clear” will remove all cases from the case panel.
- “Done?” will inform you either how many cases are absent from the case panel or whether all cases are present.
- “List” will list all the cases that have yet to be generated, with preset attempts stored in them.

Certain buttons will be visible at different times, depending on whether an action is legal at that moment in time. Also note that if you choose a new  $m$  value, then all cases will be removed from the panel. This is due to the fact that the stored attempts may no longer be valid with a new  $w$  string. One should also note that if a case example does not generate a valid partition, then all examples of that case will not necessarily do likewise. It is possible, but not for every language, so one must use intuition before generalizing one example to an entire case.

Below is the screen for when all eleven cases in this language are present in the case panel. The file with this screen present is stored in [cfUserFirst.jff](#).



## Computer Goes First

Context-free pumping lemmas when the computer goes first have similar functionality to the corresponding regular pumping lemma mode, except with a  $uvxyz$  decomposition. No cases are used for when the computer goes first, as it is rarely optimal for the computer to choose a decomposition based on cases. Thus, the case panel will never be present.

## 20.Transform Grammar

### Contents

#### Definition

#### How to Transform a Context-Free Grammar into CNF

- Remove Lambda Productions
- Remove Unit Productions
- Remove Useless Productions
- Convert to CNF

### Definition

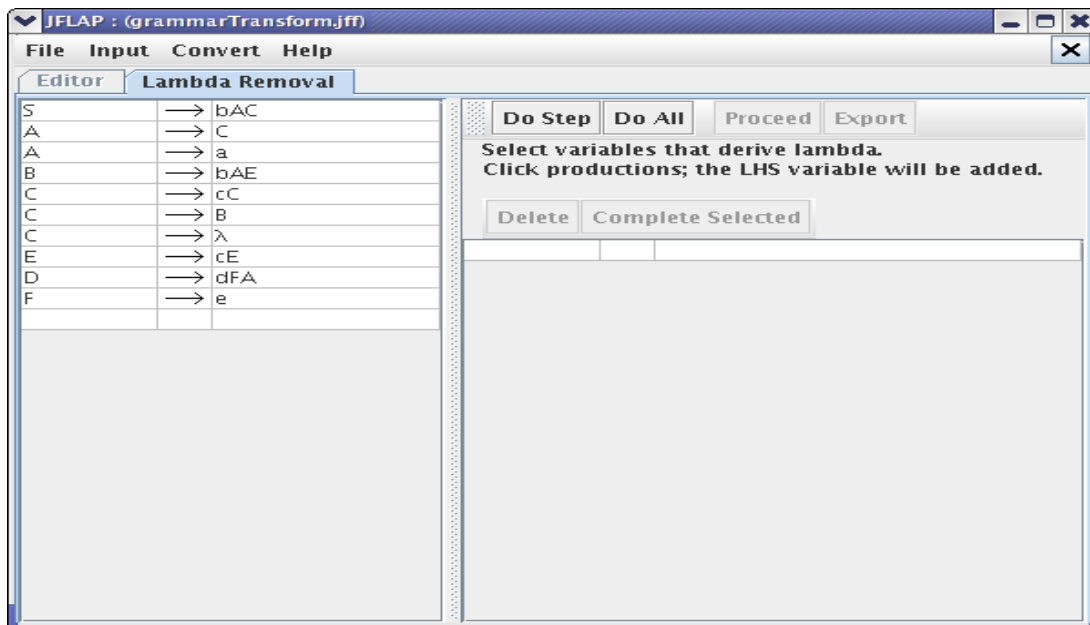
The Chomsky Normal Form (CNF) for a context-free grammar has a restricted format, in that the right-side of a production has either two variables or one terminal. CNF's restrictions result in many efficient algorithms, such as improving speed in Brute Force parsing.

#### **How to Transform a Context-Free Grammar into CNF**

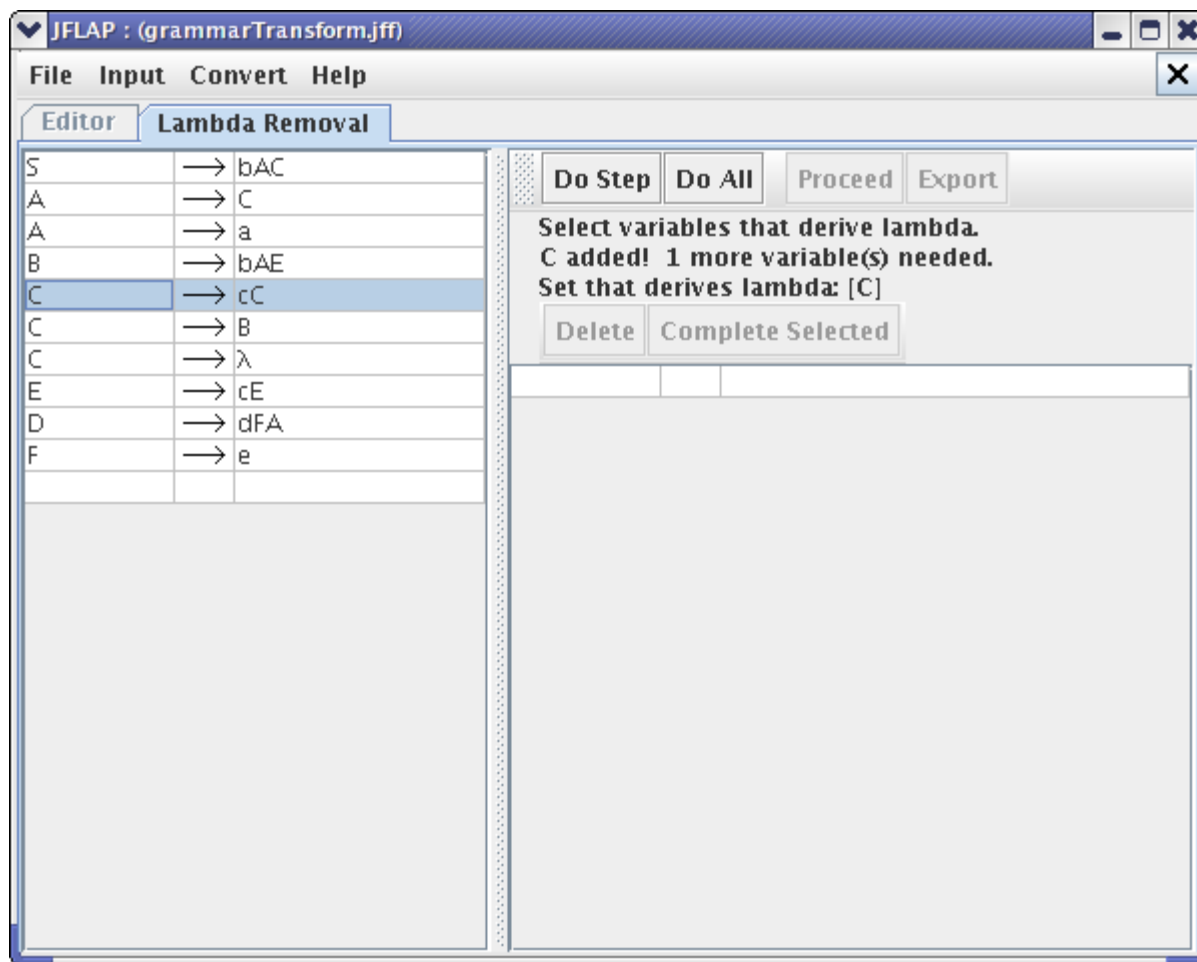
We will begin by loading the grammar in the file [grammarTransform.jff](#). Then click on the **Convert** menu, then click on **Transform Grammar**. Now, there are four steps that we have to go through in order to transform the grammar into CNF. They are removing lambda, unit, and useless productions, followed by a final step. Each step will be discussed in the following sections.

#### **Remove Lambda Productions**

After clicking **Transform Grammar**, JFLAP will open the tab **Lambda Removal** and you should be able to see a window similar to this.

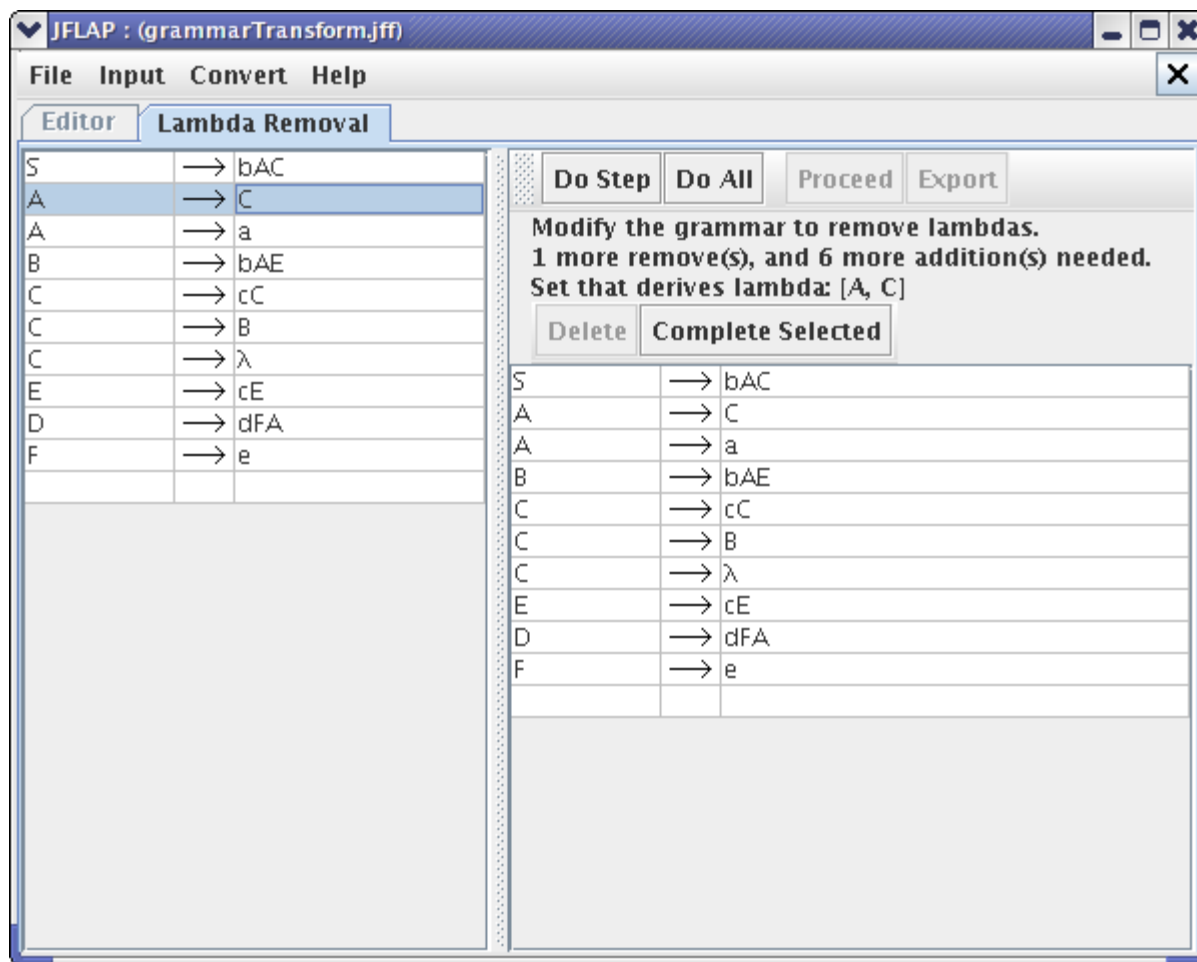


The first step is to identify the variables that can derive a lambda production. It is fairly obvious to see that the variable “C” can derive lambda via the production “C->lambda” as it is a lambda production. Therefore, we should add the variable “C” to the set of variables that derive lambda by clicking on any “C” production. After clicking, your window should look like this :

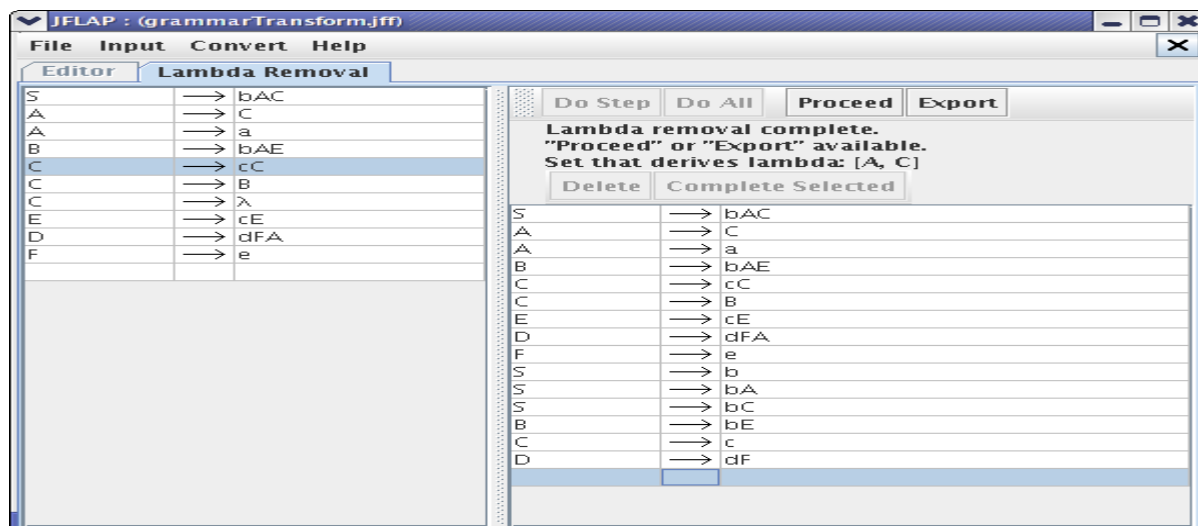


Now JFLAP tells us that there is one more variable that we need to identify. Notice that there is production "A→C". Since variable "C" could derive lambda, it can be concluded that "A→C" production could lead to a lambda production. Thus, we click on this production to add "A" to the set of variables that derive lambda. Your window should now look like this :





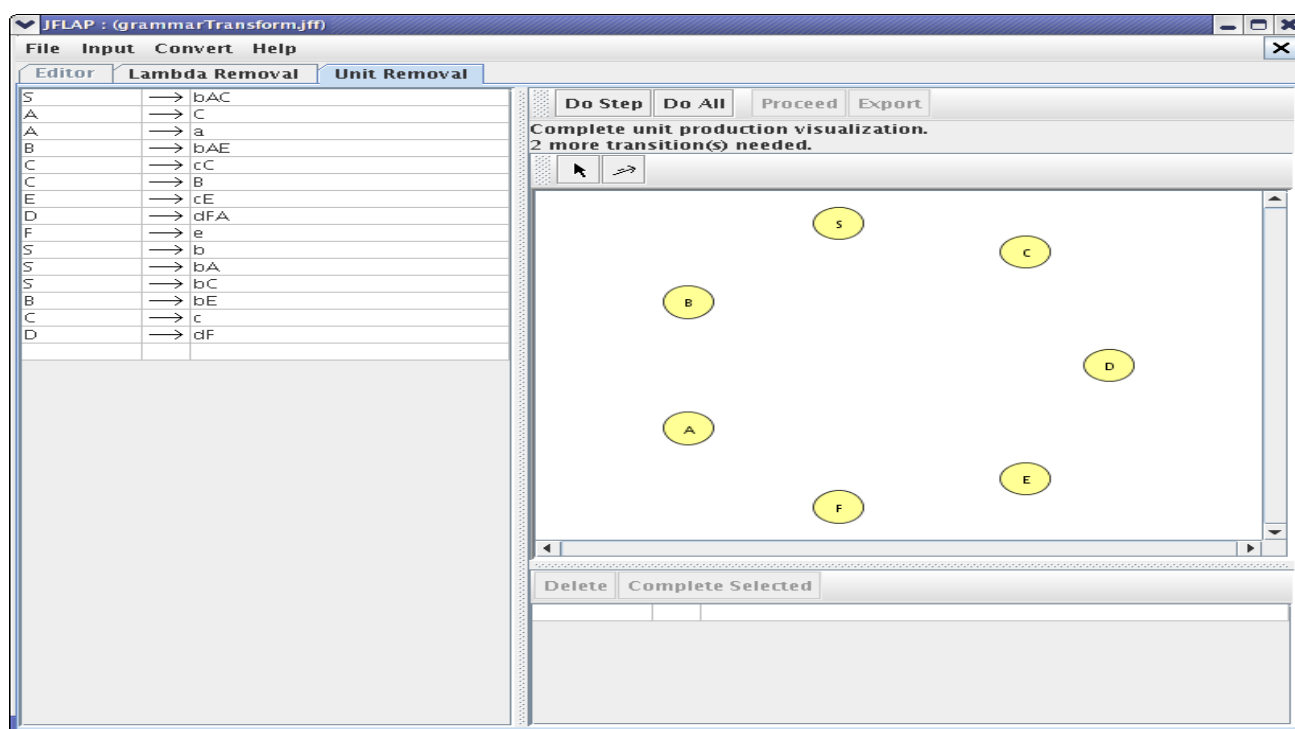
We have successfully identified all the variables that can derive lambda. Note that a copy of the grammar appears on the right-side of the window for us to modify. Get rid of all lambda productions by clicking on them and clicking **Delete**. There is one. Now, we have to add new productions to make sure that our grammar still accepts/rejects the same strings as it did before the lambda removal. JFLAP notifies you to add 6 more productions. A new production(s) must be created for productions with right-hand sides that contain a variable that could derive lambda, it could have disappeared before. However, do not add any lambda productions. You can click on the production on the left side and click **Complete Selected** to let JFLAP add new productions related to that production. You can also manually add all the productions. For instance, since variable “C” could derive lambda, we could have derived terminal “c” from variable “C” (“C- $\rightarrow$ cC” and derived C- $\rightarrow$ lambda). However, since we do not have lambda any more we have to add production “C- $\rightarrow$ c” into our grammar. After entering all the productions, your window should look like :



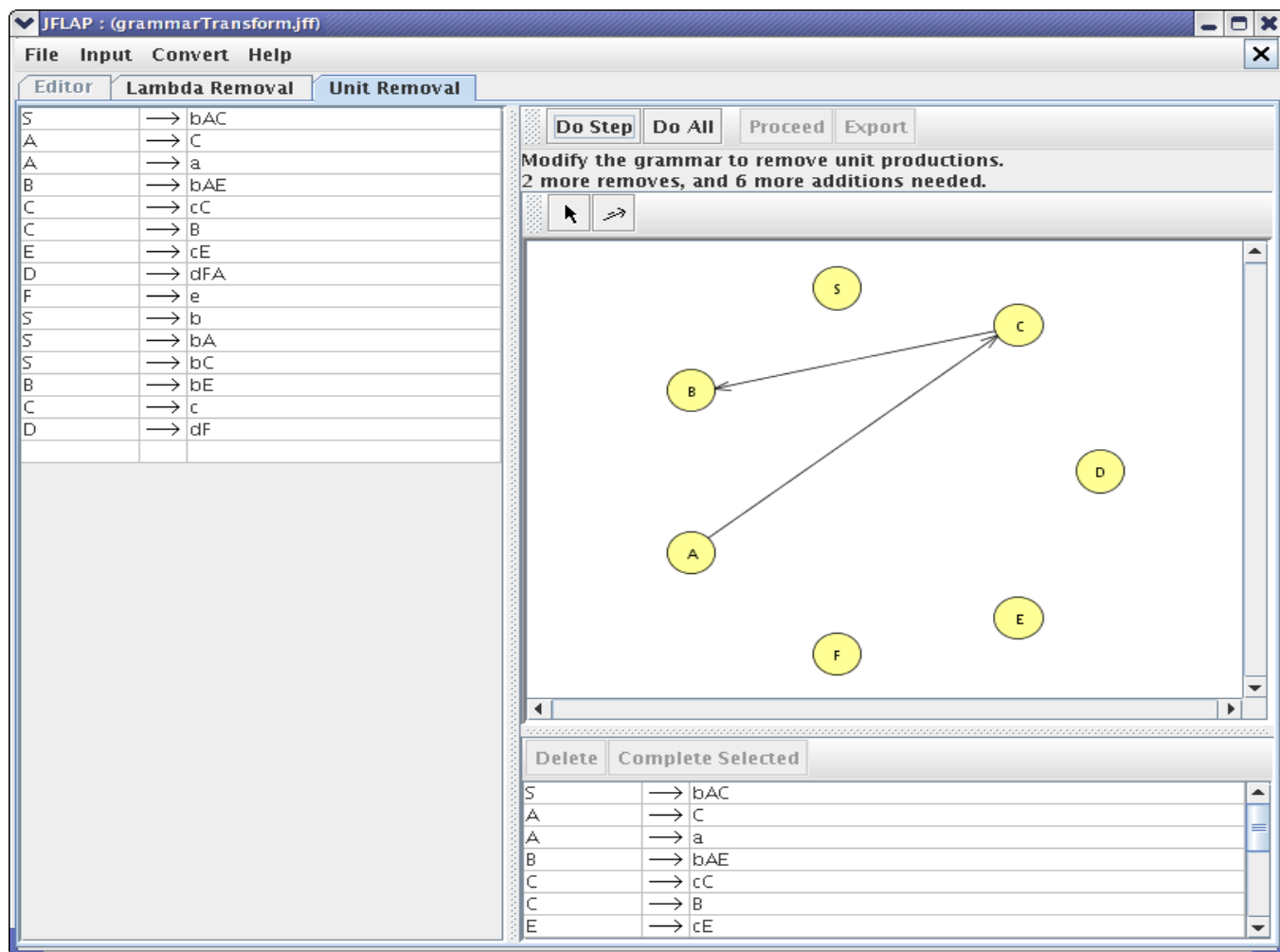
Since lambda removal is complete, we can click **Proceed** and move on to the next step.

### Remove Unit Productions

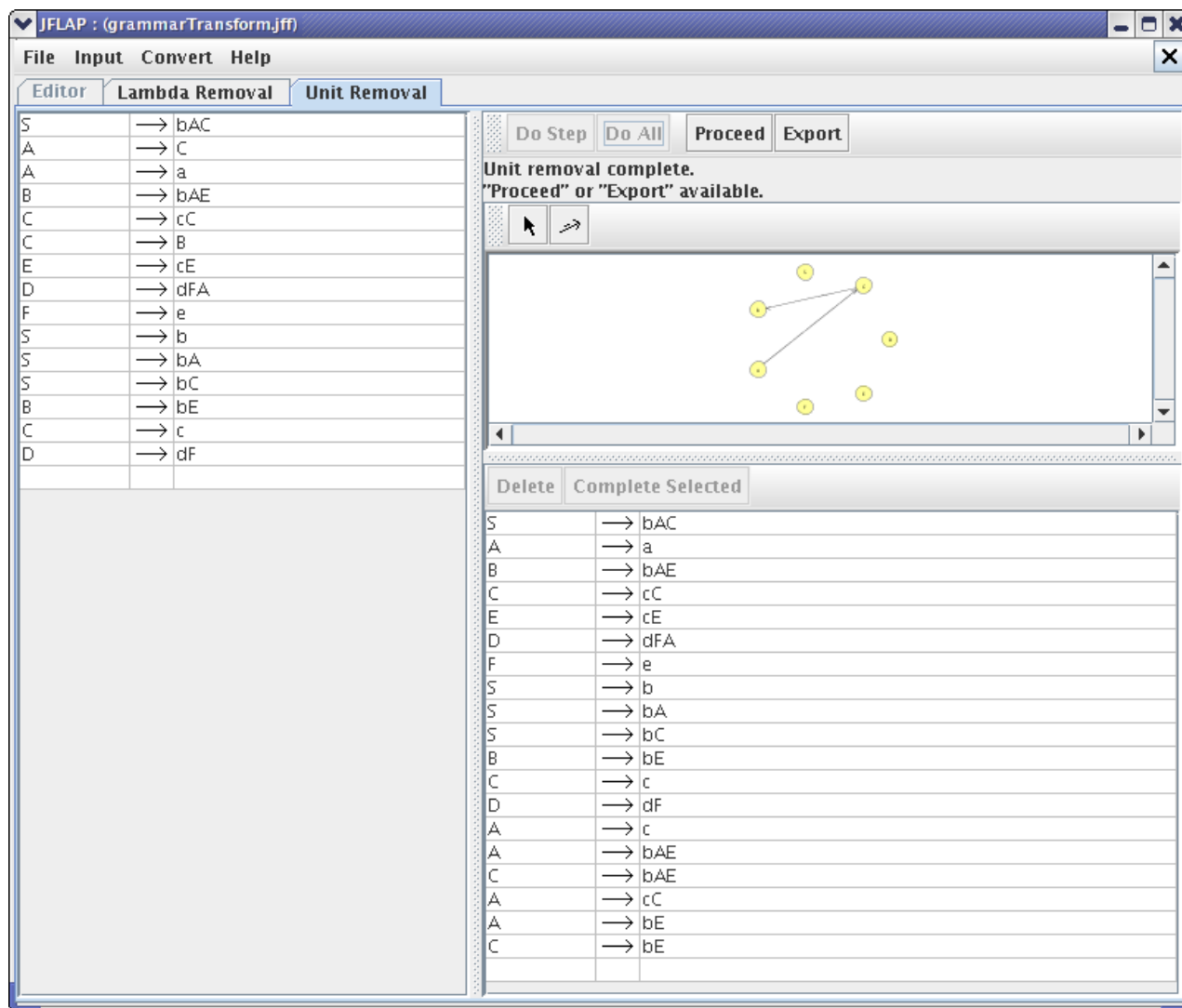
Now it's time to remove unit productions. Your unit removal window would look like this :



There is a picture with each variable from the grammar shown as a node in a graph with no edges. We will use the graph to see relationships between variables in unit productions. For each unit production "X->Y" add a directed edge(or transition) from node X to Node Y. JFLAP notifies you that you need to add 2 more transitions to the unit production visualization. The variable "A" is directly connected to variable "C" through the production "A->C". For the similar reason, we can connect variable "C" to "B". After we finish adding these 2 transitions, our window should look like :



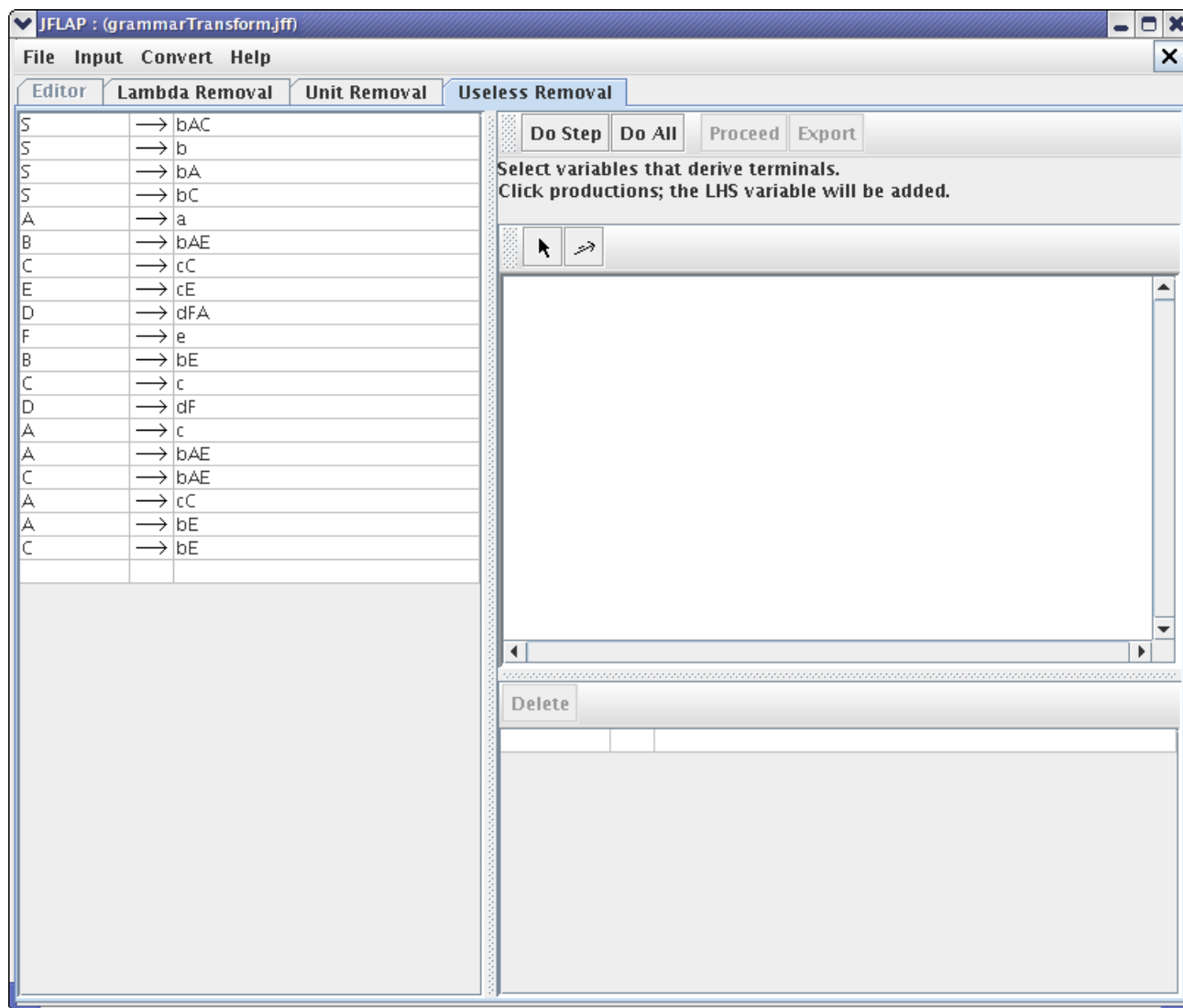
A copy of the grammar you can modify is now shown in the lower right window. First remove the unit productions by selecting them in the bottom right panel. Select the production and click **DELETE**. After the deletion is complete, we need to add new productions. For unit productions, both implicit and explicit (such as “A→B” from “A→C” and “C→B” as shown from the visualization), additional rules must be added to handle the missing unit productions. We have to make sure this changed grammar still accepts/rejects the same strings as it did before. For example, since we removed the production from variable “A” to variable “C”, we cannot derive terminal “c” from variable “A”. So, we have to add the production “A→c” into our grammar. JFLAP kindly notifies us how many more productions we have to add. After we finish adding all the productions, we are finished with unit removal and the window should look like this :



Now, we click on **Proceed** button and move on to the next step.

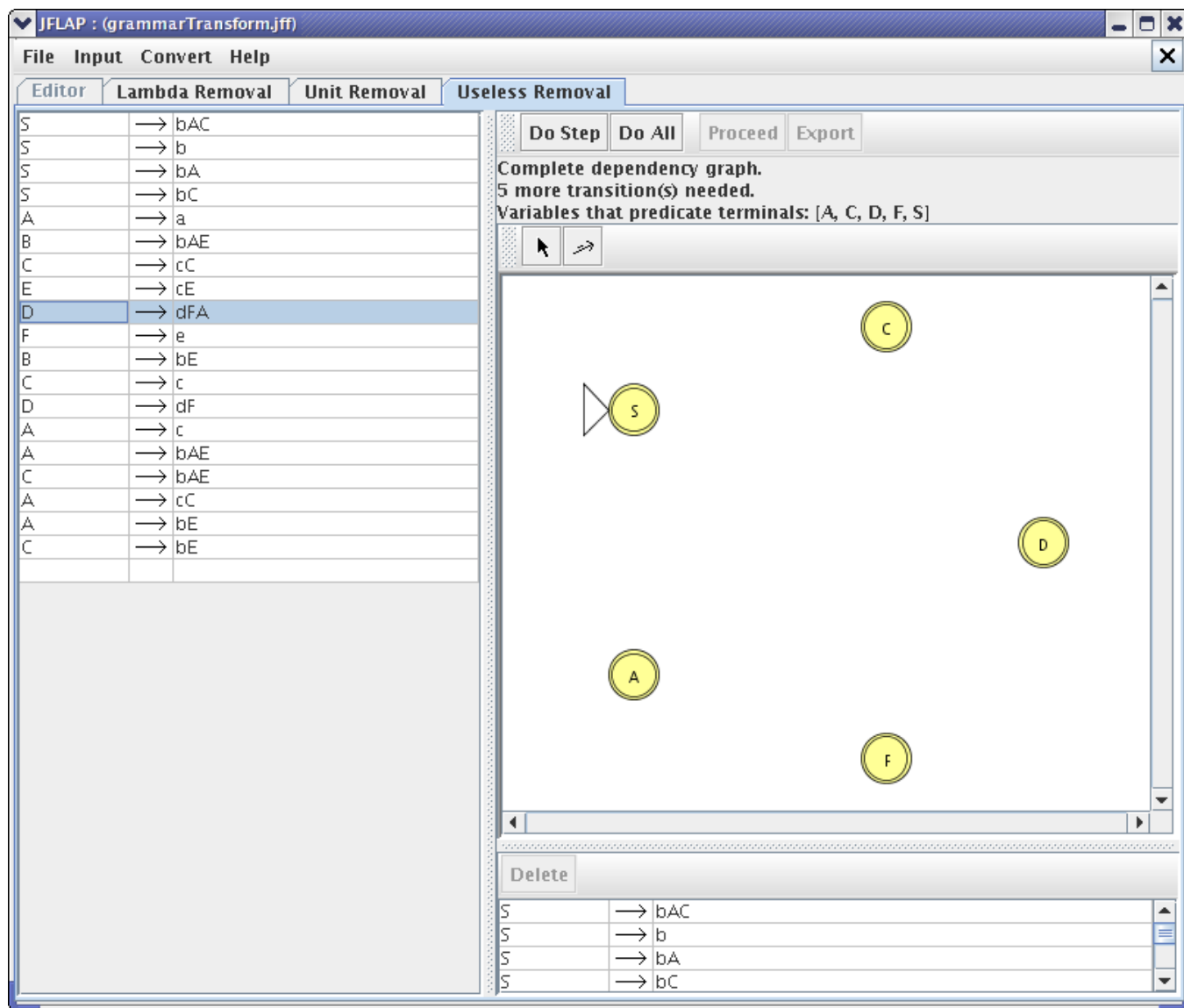
### Remove Useless Productions

After successfully removing both lambda and unit productions, your grammar window would look like :



A production is useless if no derivation can use it. Removing useless productions is a two-step process. First, find useless variables that cannot derive any string of terminals and remove productions with those variables. Second, find variables that are unreachable from the start symbol and remove productions with those variables.

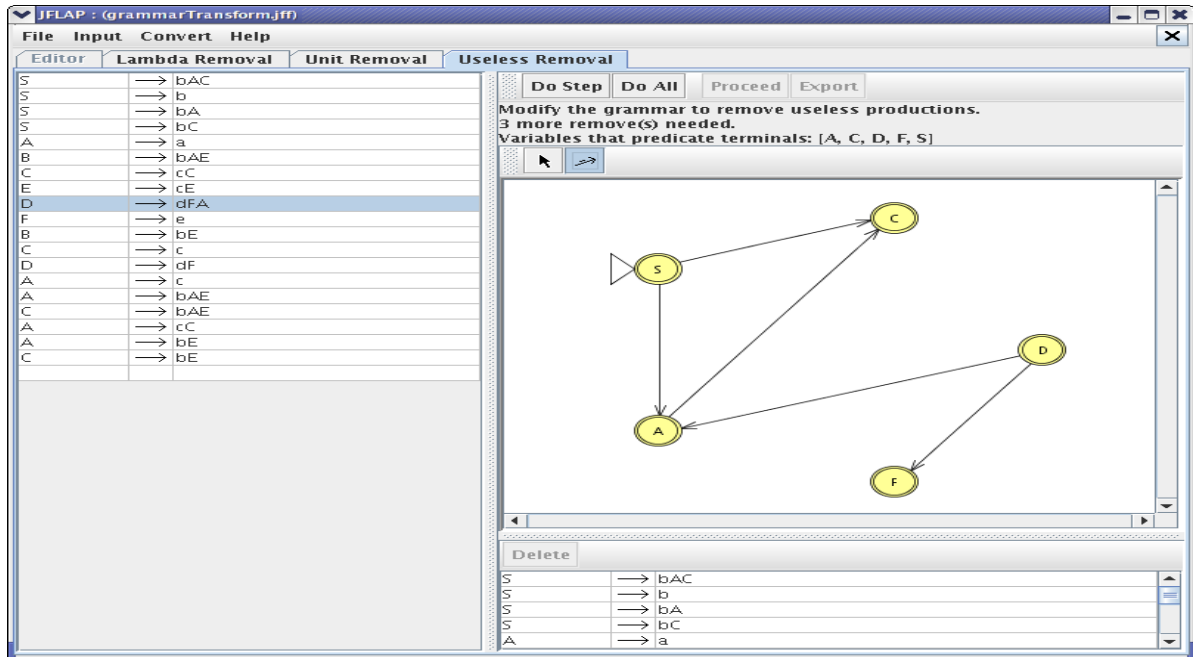
As JFLAP indicates, first we click on variables that derive terminals directly. They are variables “S”, “A”, “C”, and “F”. Also, if there are productions with only these variables on the right-side, then the variable on the left-side can also derive a string of terminals, and should be added such as “D” from “D→dF”. Click on variable “D” to add it. After clicking these variables, your window should look like :



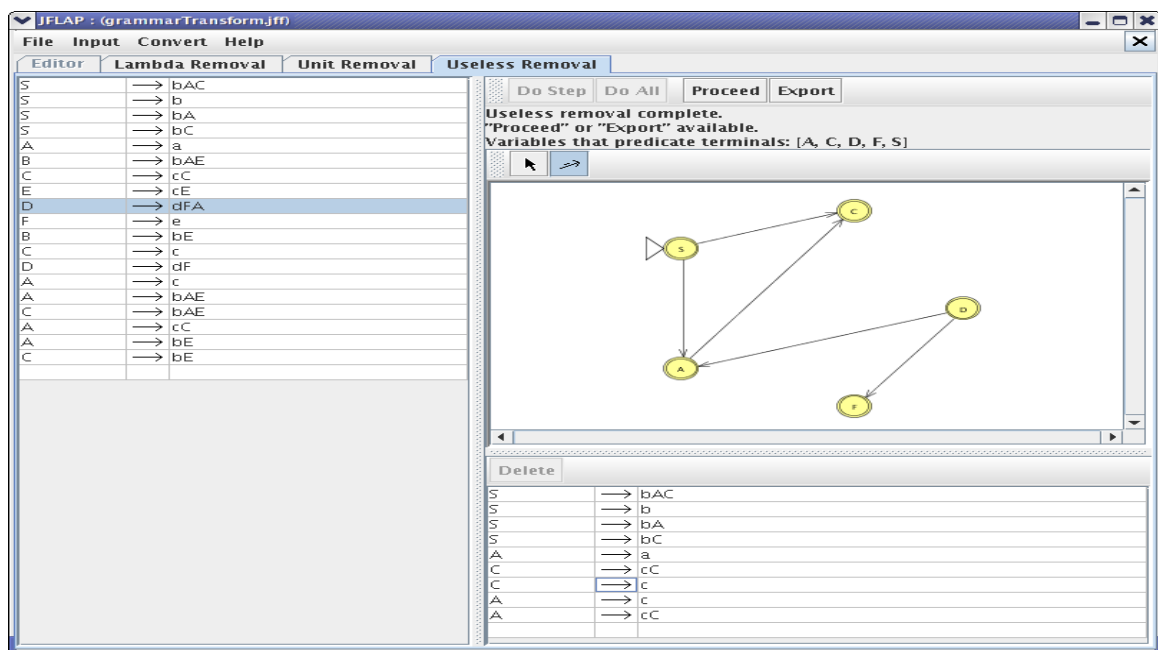
There are two new items shown. A visualization graph and below it, a copy of the grammar is shown, with rules missing that have variables that could not derive a string of terminals, productions with the variables “B” and “E”.

The visualization has a node for each variable in the remaining grammar. This graph is different then the graph for removing unit-productions. For this graph, add a transition from variable “X” to variable “Y” if there is a production with “X” on the left-side and “Y” anywhere on the right-side.

Now, we have to add 5 dependions among these variables. From the production “S->bAC”, we know that variable “S” depends on both “A” and “C”. We also know that “A” depends on “C” based on the production “A->cC”. We will add these transitions to our state diagram, by clicking the arrow in the panel and creating transitions. Add the remaining transitions. Now your window would look like :



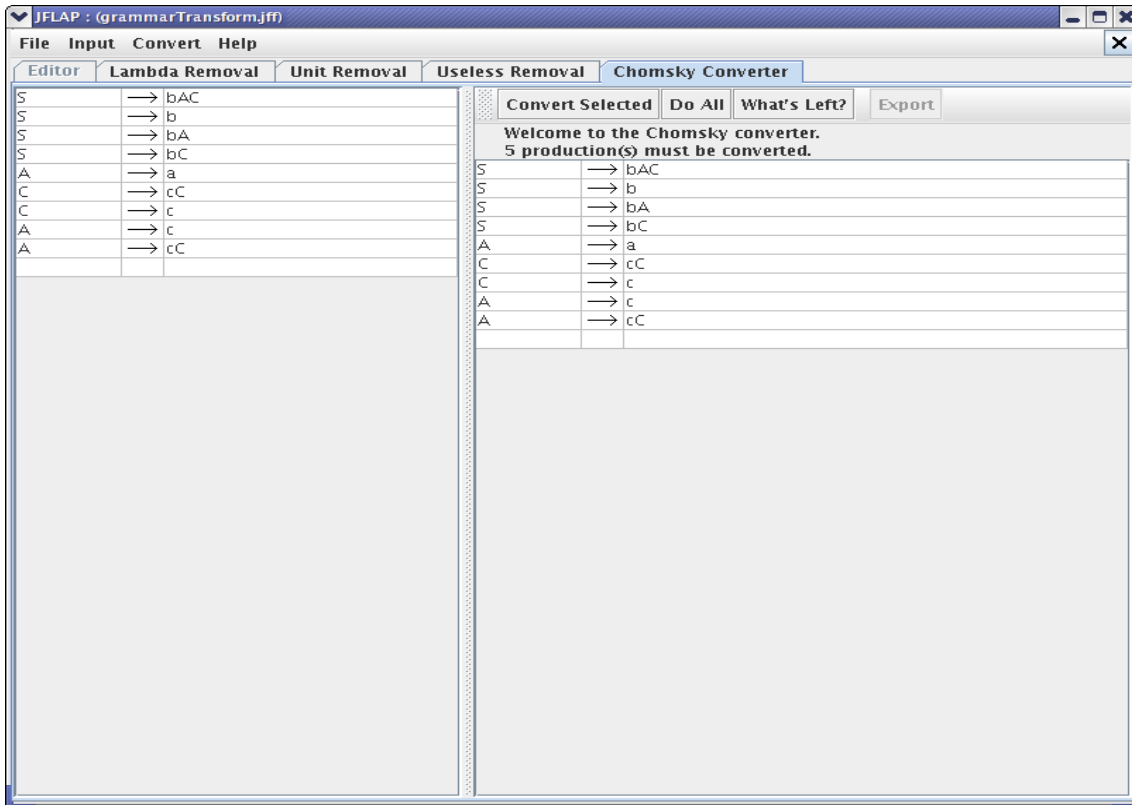
We now need to remove rules with those variables that are not reachable from “S”. That would be rules with “D” and “F” as those nodes are not reachable from “S” in the graph. Click on them and then click **DELETE**. After eliminating all of the useless productions, your window should look like :



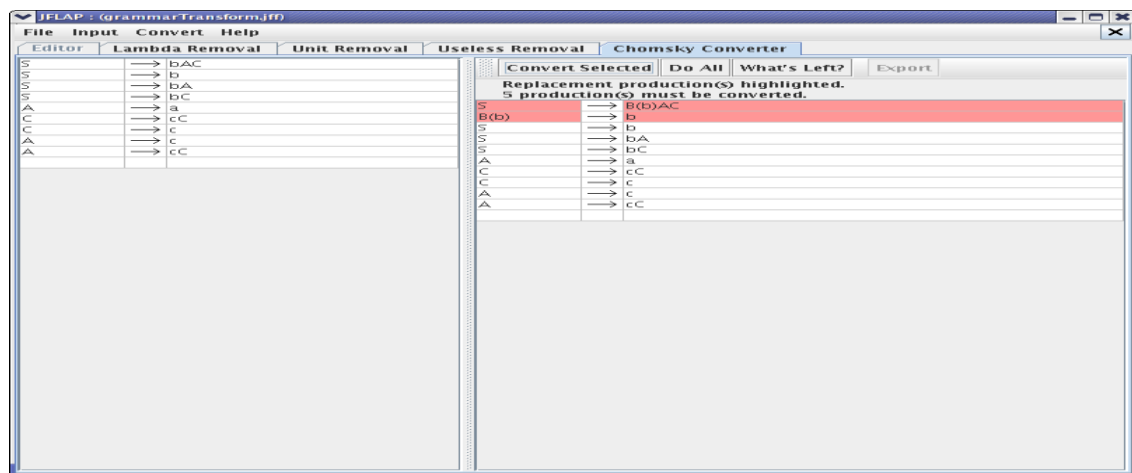
Now we have successfully removed useless productions. By clicking on the **Proceed** button, we will move on to our final step.

### Convert to CNF

Now, we are at the final step of converting our grammar to CNF. In CNF, the right side of a production is either one terminal or two variables. If you have done everything correctly in previous steps, your window should look like this now :

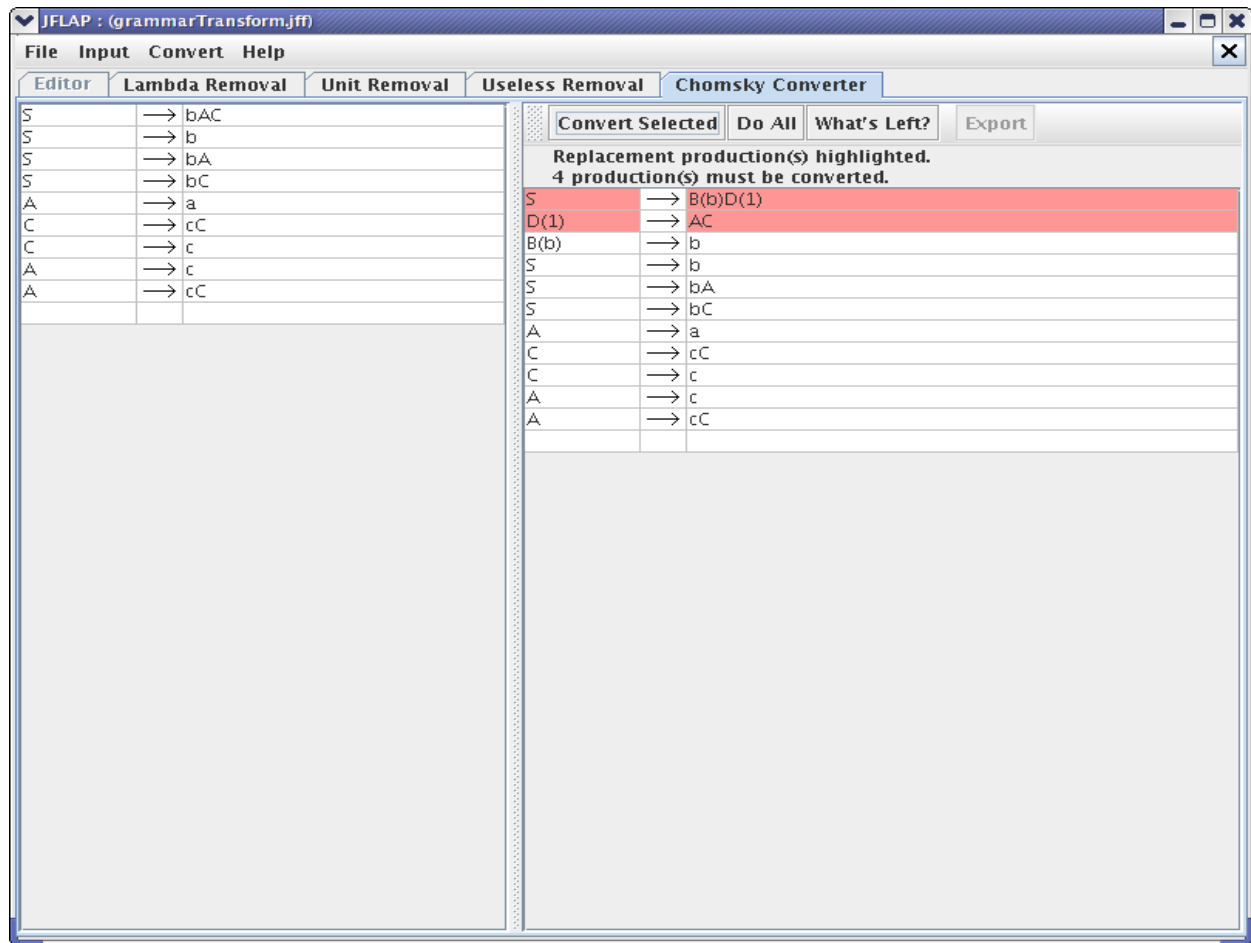


Notice that the production “ $S \rightarrow bAC$ ” does not follow our CNF restriction and it must be converted. You can convert this production by clicking on the production on the right panel and click **Convert Selected**. Now your JFLAP window will look like :



We have replaced the terminal “b” in “ $S \rightarrow bAC$ ” with a new variable “B(b)” and a production “ $B(b) \rightarrow b$ ”. The production “ $S \rightarrow B(b)AC$ ” now has all variables on the right-side, but has too many. Clicking **Convert Selected** again expands this production into 2 productions “ $S \rightarrow B(b)D(1)$ ” and “ $D(1) \rightarrow AC$ ” and another new variable “D(1)”. After following this step, your window should look like :





As JFLAP indicates, we have to convert 4 more productions. Try to select those productions, and create the missing productions.

Alternatively, click on **Do All** to finish the conversion. After we are finished with the conversion, the grammar will be in CNF. Our final window should look like below and we can export this grammar to utilize in fast parsing.

