

# JFLAP -Java Formal Language and Automata Package

Working with Finite Automata

2009-2010



# **JFLAP**

## **(Java Formal Language and Automata Package)**

### **1. Introduction: Creating your Finite Automata**

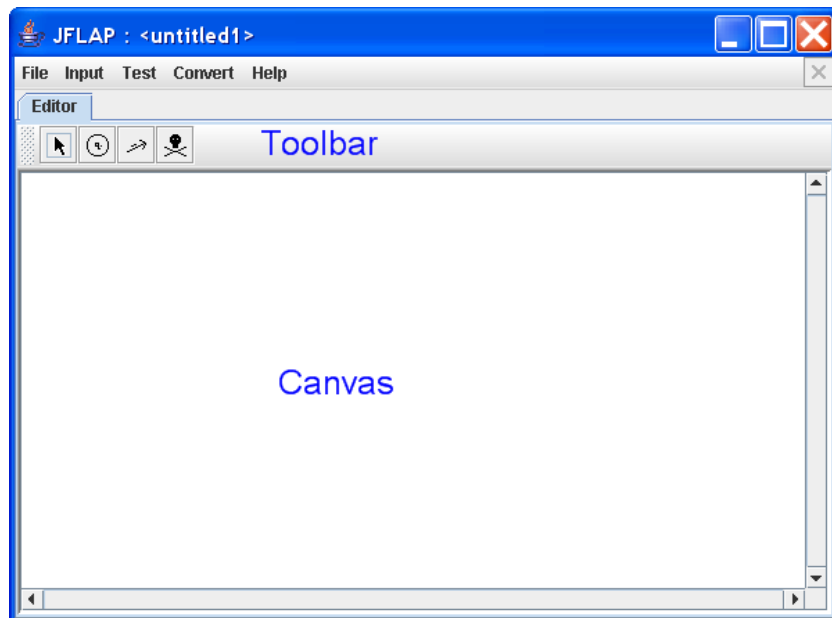
JFLAP defines a finite automaton (FA)  $M$  as the quintuple  $M = (Q, \Sigma, \delta, q_s, F)$  where

- $Q$  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

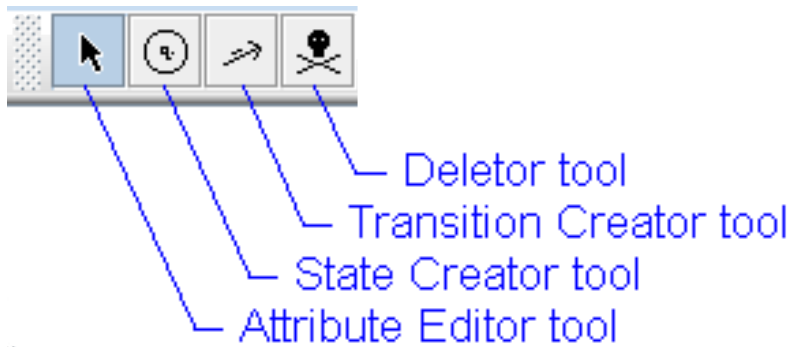
Note that this definition includes both deterministic finite automata (DFAs), which we will be discussing shortly, and nondeterministic finite automata (NFAs), which we will touch on later.

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.

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



Let's take a closer look at the 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

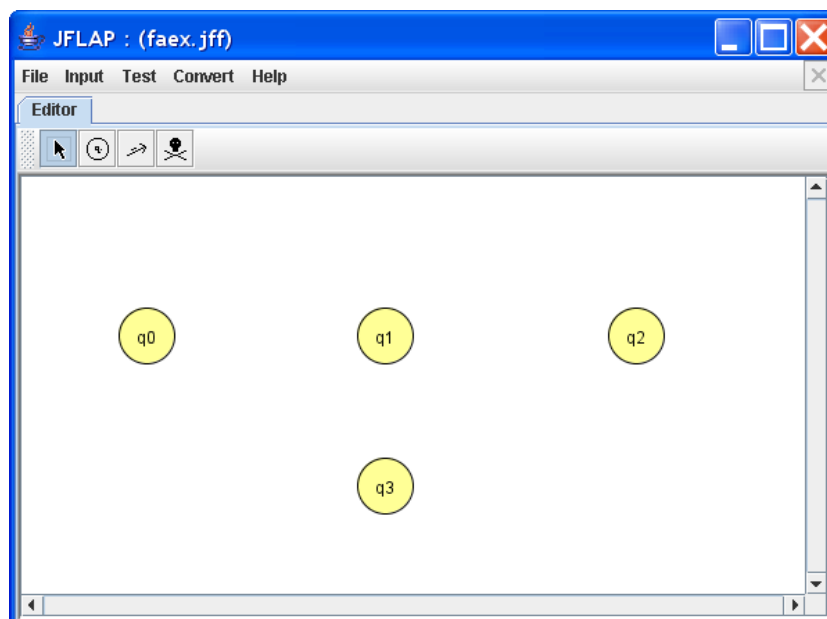
To select a tool, click on the corresponding icon with your mouse. When a tool is selected, it is shaded, as the Attribute Editor tool is above. Selecting the tool puts you in the corresponding mode. For instance, with the toolbar above, we are now in the Attribute Editor mode.

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.


### 1.1. **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:



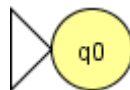
Now that we have created our states, let's define initial and final state.

### 1.2. 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:



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.



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.



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

### 1.3. Creating Transitions

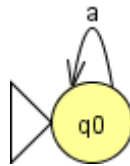
We know strings in our language can start with  $a$ 's, so, the initial state must have an outgoing transition on  $a$ . We also know that it can start with any number of  $a$ 's, which means that the FA should be in the same state after processing input of any number of  $a$ 's. Thus, the outgoing transition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

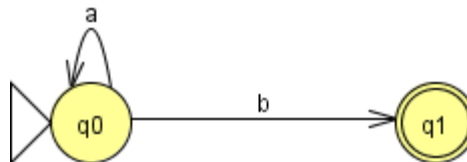


Note that  $\lambda$ , representing the empty string, is initially filled in for you. If you prefer  $\epsilon$  representing the empty string, select **Preferences: Preferences** in the main menu to change the symbol representing the empty string.

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:



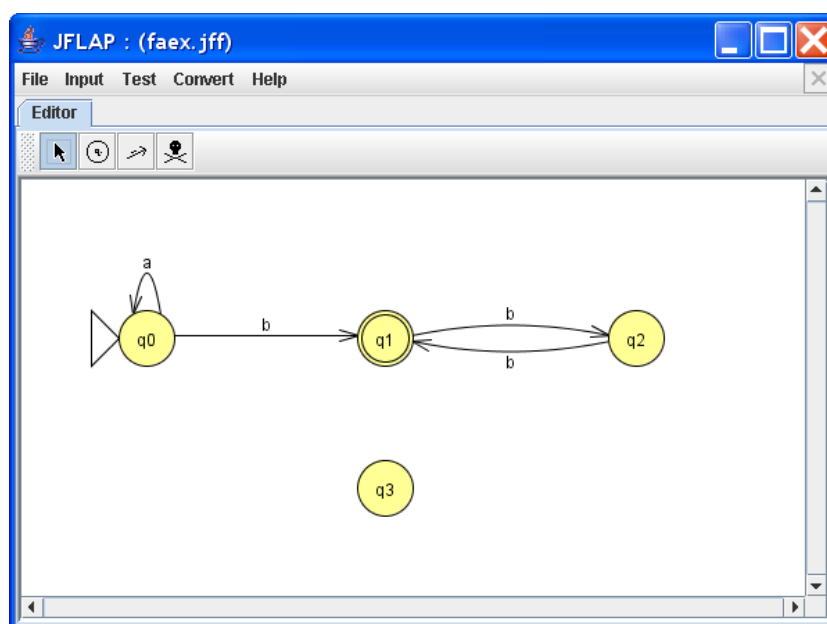
Next, we know that strings in our language must end with a odd number of  $b$ 's. Thus, we know that the outgoing transition on  $b$  from  $q_0$  must be to a final state, as a string ending with one  $b$  should be accepted. To create a transition from our initial state  $q_0$  to our final state  $q_1$ , 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:



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  $b$ , which 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  $q_2$ .

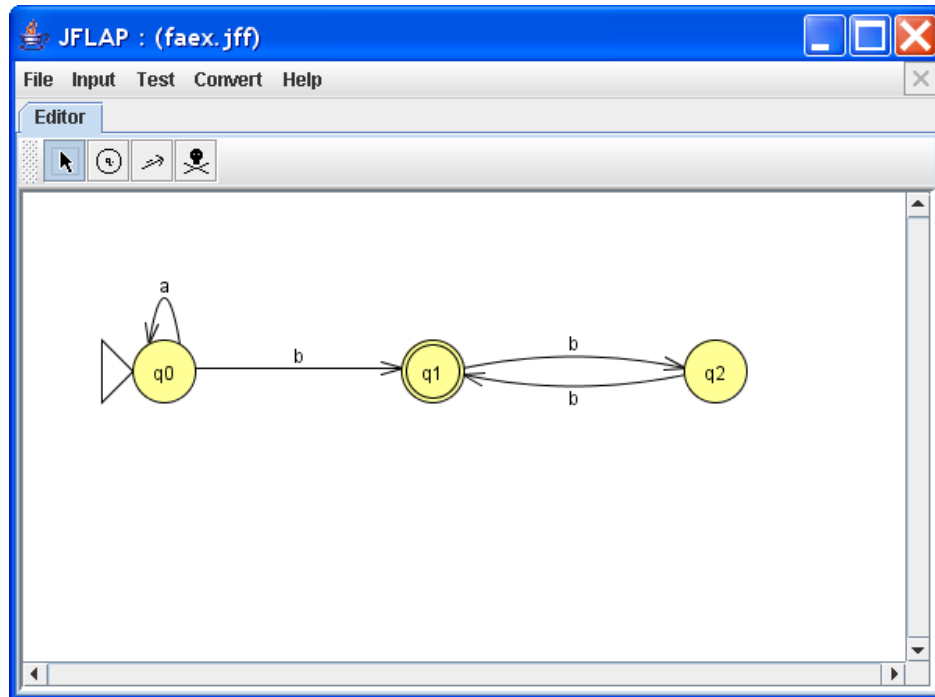
Create a transition on  $b$  from  $q_1$  to  $q_2$ . As the FA should accept strings that end with an odd number of  $b$ 's, create another transition on  $b$  from  $q_2$  to  $q_1$ . Your FA is now a full, working FA! It should look something like this:



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

#### 1.4. Deleting States and Transitions

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



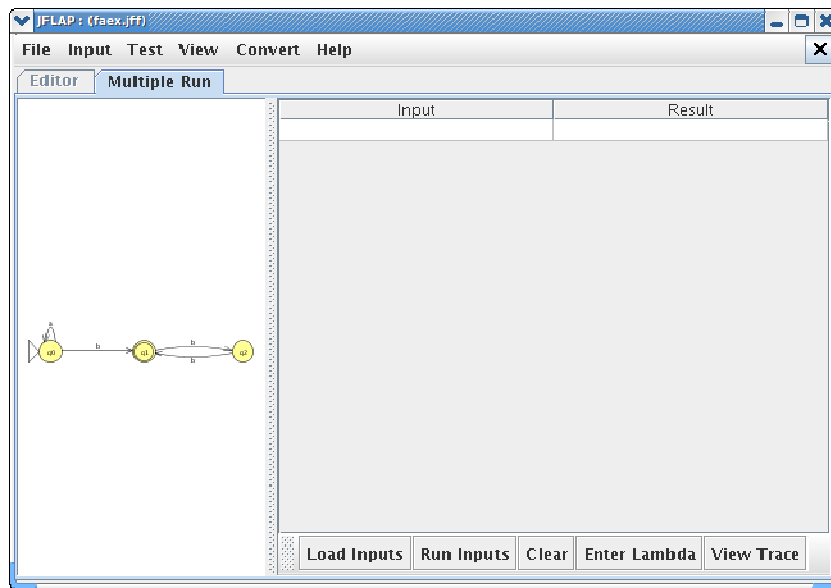
Similarly, to delete a transition, simply click on the input symbol of the transition when in Deletor mode. Your FA is now complete.

#### 2. Testing your Finite Automata: Running the FA on Multiple Strings

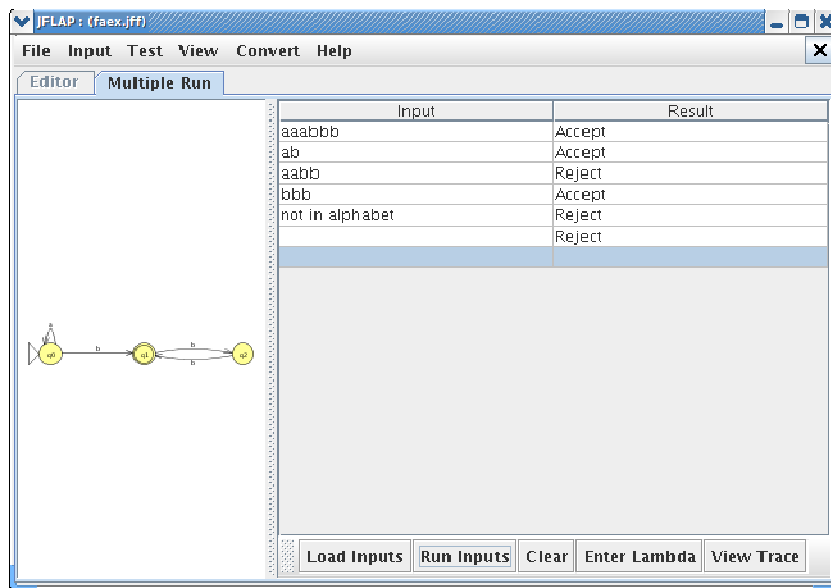
Now that you've completed your FA, you might want to test it to see if it really accepts strings from the language. To do so, select **Input: Multiple Run** from the menu bar.



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



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.

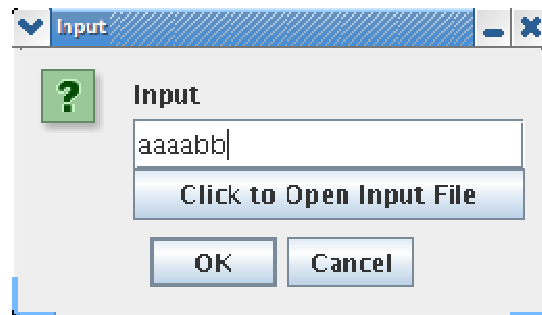


Clicking **Clear** deletes all the input strings, while **Enter Lambda** enters the empty string at the cursor. **View Trace** brings up a separate window that shows the trace of the selected input. To return to the Editor window, select **File: Dismiss Tab** from the menu bar.

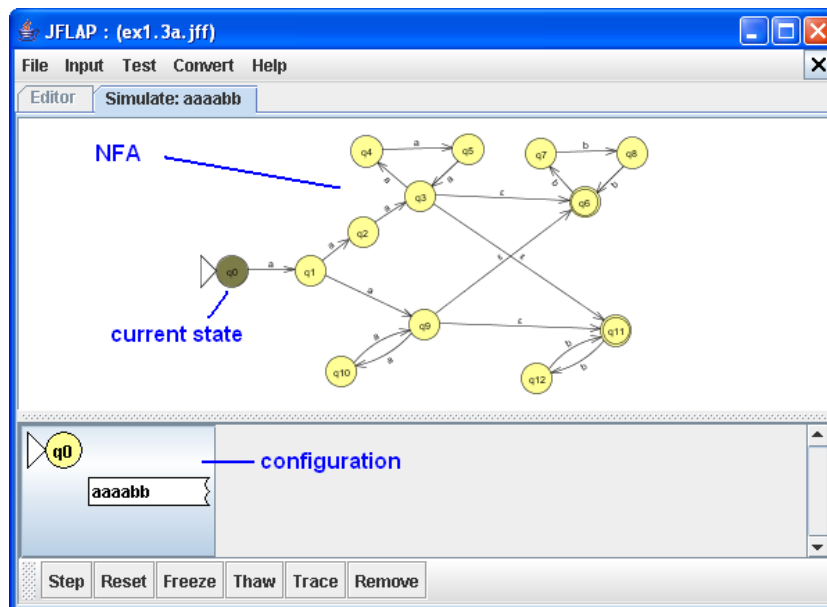
## 2.1. Running Input on an NFA

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.

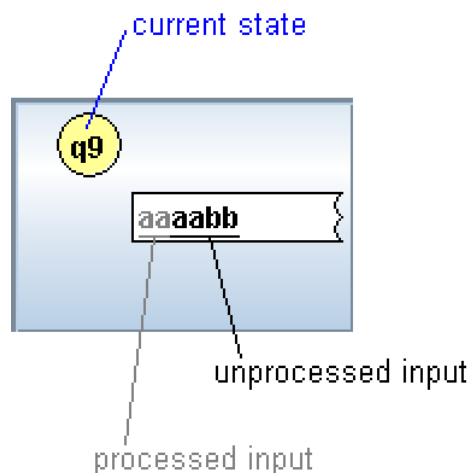
**NOTE:** When loading input from the file, JFLAP determines end of input string by the white space. Thus if there is string “ab cd” in a file, only “ab” will be considered as an input (“cd” will be ignored since there is a white space before them).



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



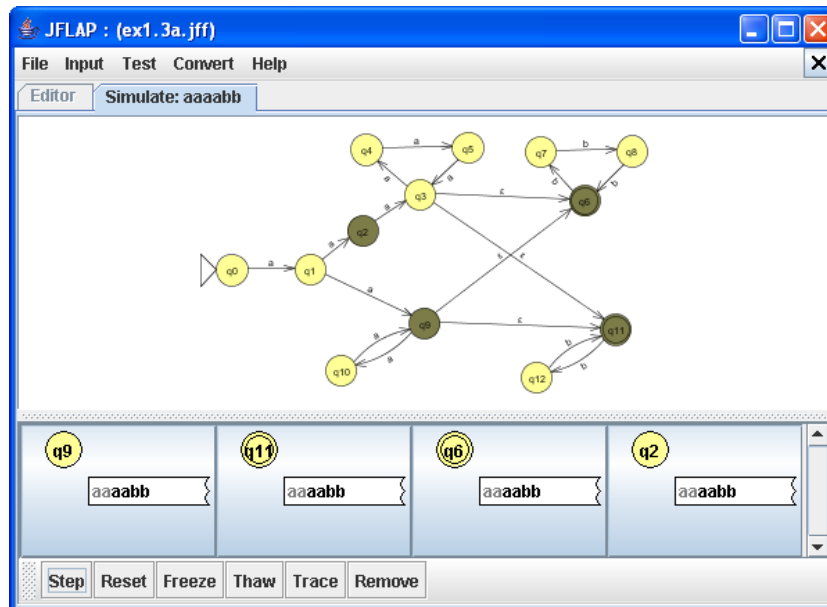
First, let's take a closer look at a configuration:



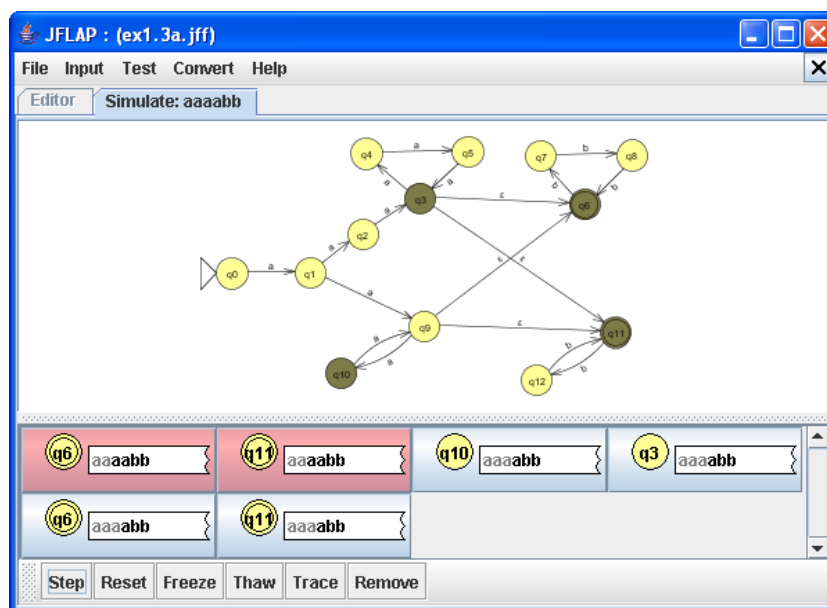


The configuration icon shows the current state of the configuration in the top left hand corner, and input on the white tape below. The processed input is displayed in gray, and the unprocessed input is black.

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.



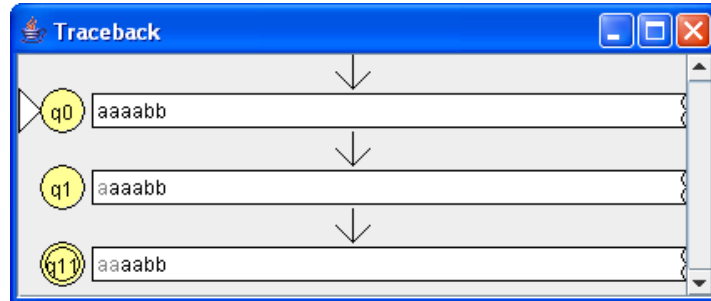
This is because the machine is nondeterministic. From  $q_1$ , the NFA took both  $a$  transitions to  $q_2$  and  $q_9$ . As  $q_9$  has two  $\lambda$ -transitions (which do not need input), the NFA further produced two more configurations by taking those transitions. Thus, the simulator now has four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



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.

## 2.2. 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:



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.

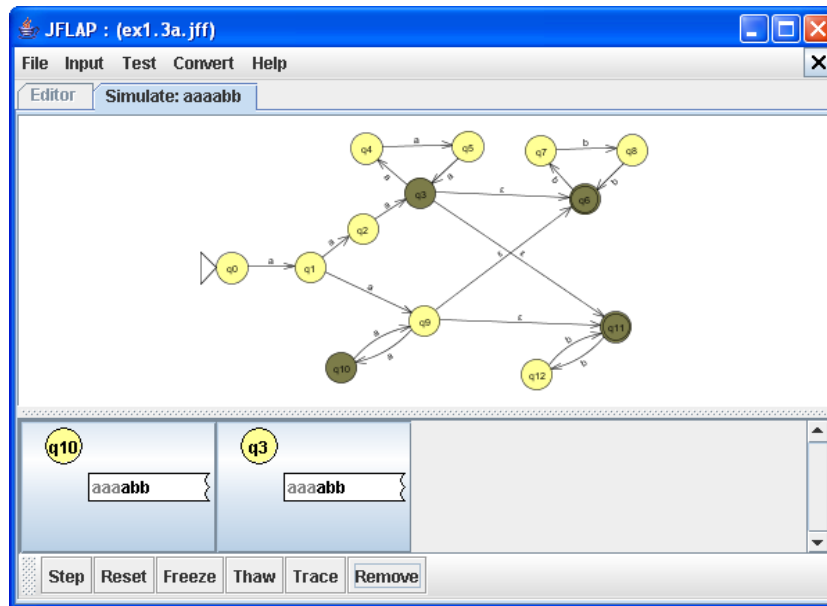
Although rejected configurations will remove themselves in the next step, we can also remove configurations that have not been rejected.

## 2.3. 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:



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.

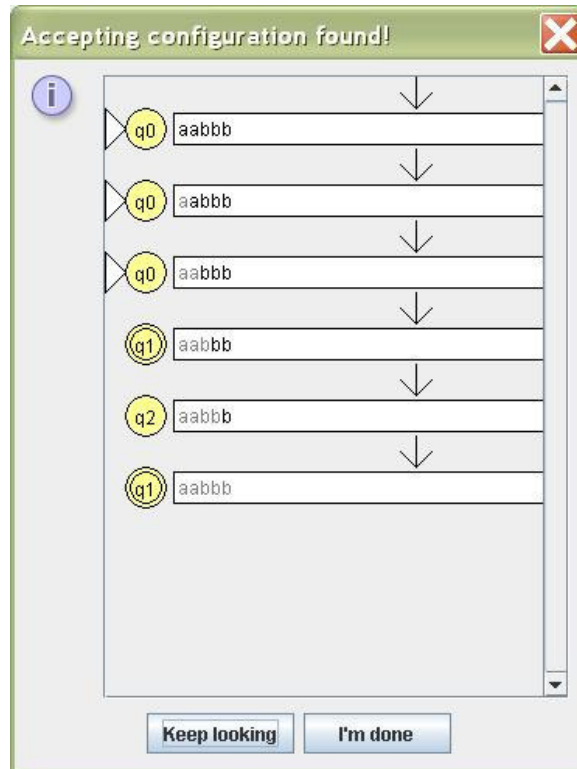
#### 2.4. Resetting the simulator

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.

#### 2.5. Fast Simulation

Using this option we can quickly verify if an automaton accepts or not an input. If this is accepted, we will see the list with the steps followed to (similar to the Trace option). If not, you will see an informative message.

You can close the window using the **I'm Done** option. We will use **Keep Looking** for the non-deterministic automata.

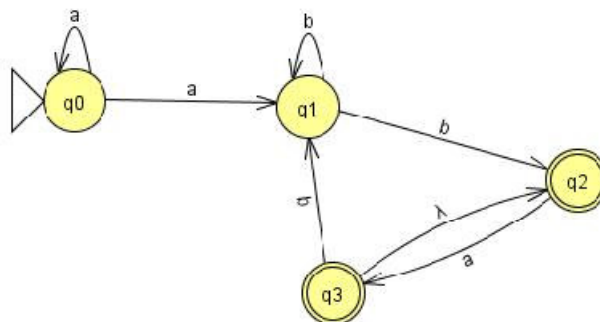


## NFA to DFA and minimal DFA


This section summarizes the process to transform a NFA into its equivalent DFA and how to reduce it to the DFA with the minimal number of states.

### *NFA to DFA*

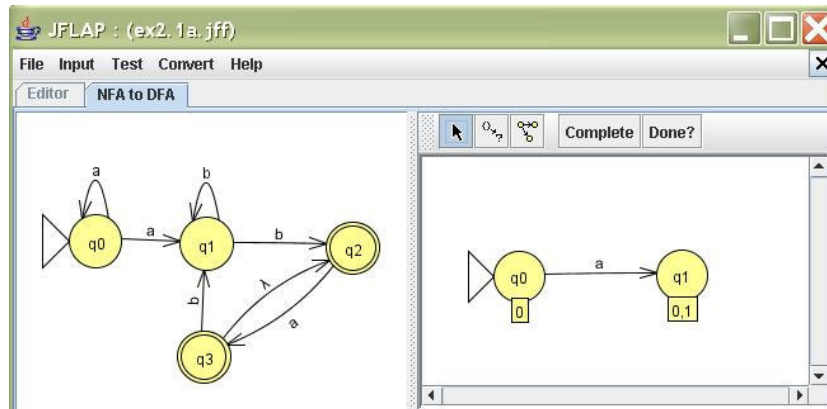
The idea is to create states in the DFA that represent a set of states of the NFA. The initial state in the DFA represents the initial state in the NFA and any accessible state from it with  $\lambda$ . For each new state in the DFA and for each symbol of the alphabet, all the accessible states from the corresponding non-deterministic state have to be determined and combine them in a new state for the DFA. Let's transform the following NFA:



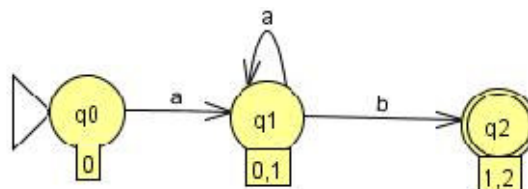
Draw the NFA and click **Convert: Convert to DFA**. The initial state in the DFA is  $q_0$  and has the label 0, which represents the state  $q_0$  of the NFA. After this we can add the

state that is reached from  $q_0$  reading  $a$ . Select the option **Expand Group**, of the tool . Click and maintain pressed the button of the mouse on the state  $q_0$ . Then, drag along the mouse towards where you want to place the following state and release the button.

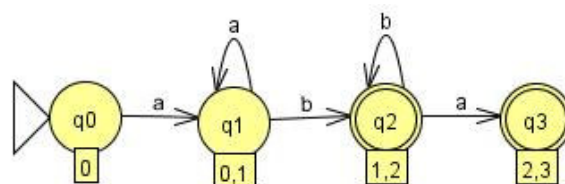
You will see a window asking for the terminal that we are going to expand (symbol  $a$ ). When introducing this, JFLAP will ask for the group of states of the NFA that are accessible from  $q_0$  when introducing one  $a$ , that they will be  $q_0$  and  $q_1$  (these states are represented by numbers 0 and 1, that is the information that is introduced in this window). Now, a new state  $q_1$  will appear:




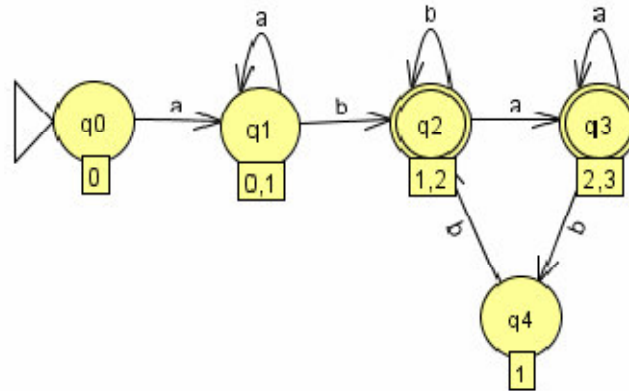
Try to expand the state  $q_0$  with the terminal  $b$ . Since there is no way in the NFA with this characteristic, a warning message will appear. Now expand the state  $q_1$  of the DFA with terminal  $a$ . Take into account that this state represents the states  $q_0$  and  $q_1$  of the NFA. In the NFA, using the terminal  $a$ , from the state  $q_0$  we transit to  $q_0$  and  $q_1$  has not transitions using this terminal symbol. The union of this two states results  $(0,1)$ , that will be  $q_1$  of the DFA. For it, expand  $q_1$  by adding a loop. After that, expand it again in the state  $q_1$  with  $b$ . The result of these expansions is:



The state  $q_2$  of the DFA is shown a final state, because the equivalent states in the NFA are also final states. Expand now the state  $q_2$  of the AFD with  $a$ . This state is represented by the states  $q_1$  and  $q_2$  of the NFA, in which  $q_1$  does not have transitions with  $a$ , and  $q_2$  has it. Expand now the state  $q_2$  of the DFA with  $b$ . The resulting DFA is the following:



There is another form to expand a state, by using the tool . When it is selected and you click on a state, all the arcs that leaves from this state automatically appear. Test this in q3. Is this the complete automaton? Pressing the button **Done**, JFLAP will inform us about elements that still lack and it will show the complete set of arcs. In our case, it even lacks a transition, for the new state q4. Use the previous tool to complete it, and then obtain the following automata:



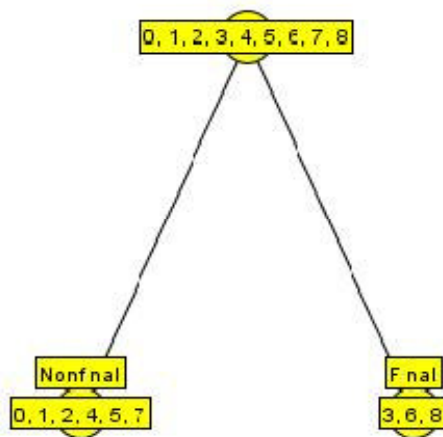
Now press again the button **Done**. The complete AFD is then exported to a new window. You can also press the button **Complete** at any time and the construction process of the DFA will be automatically completed. The new DFA must be equivalent to the NFA. To prove this, select the bar **Test: Compare Equivalence**.

### ***Obtaining the minimal DFA***

In this section, it is described how to transform a DFA into an equivalent automaton with a minimum number of states. The main idea is to consider two states  $p$  and  $q$  of a DFA that process a symbol from their state. If there is at least one word  $w$  for which the states  $p$  and  $q$  process it, and one of the states accepts  $w$  and the second one rejects  $w$ , then those states can be distinguished and they cannot be combined. On the other hand, if the states  $p$  and  $q$  act in the same way, it means that they are indistinguishable, and can be combined.

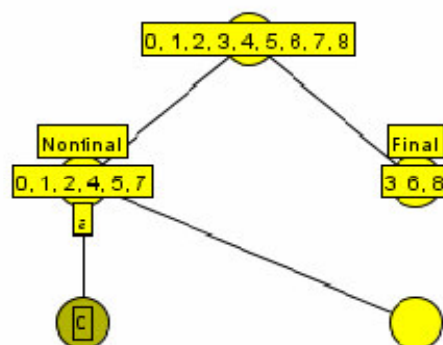
Open the file in which you have saved the automaton showed in the last figure of previous section and select **Convert: Minimize DFA**. The window will be divided in two showing the DFA (left) and a tree with states (right). At the beginning we assume that all the states are indistinguishable (the root of the tree contains all the states). Once we determine a new distinction between states, we will add a node in the tree to show it. The process continues until there are not more possible divisions. Each leaf of the final tree will represent a group of indistinguishable states.

The first step that JFLAP will show in this window is for distinguishing final and nonfinal states. For this reason, the tree is divided showing the set of final states and the nonfinal ones:

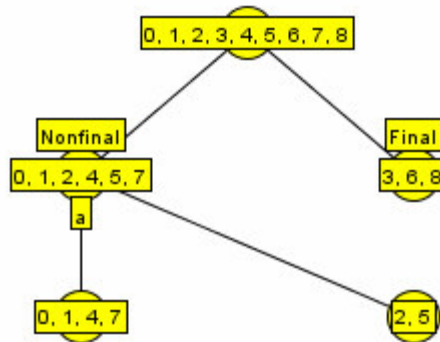


For future divisions, a terminal is selected that distinguishes the states in the node. If some of the states in a node leaf, for that terminal, go to some state of another node leaf, and other states with the same terminal go to states that are in another node leaf, then the node will have to be divided in two groups of states. Let's explore the first node leaf with the nonfinal states. What happens for each one of those states if we processed a  $b$ ? From the state  $q_0$  we transit to  $q_2$ , from  $q_1$  to  $q_0$ , and so on. Each one of those states goes to a same state of that node. Therefore, the input  $b$  does not distinguish them. If you try to introduce  $b$ , in **Set Terminal**, JFLAP will show you a message informing about this.

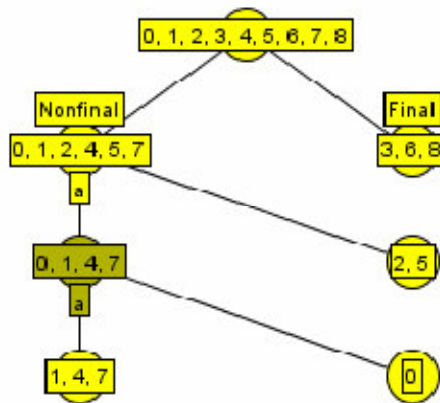
Select again **Set Terminal** and introduce the terminal  $a$ . In this case, this symbol distinguishes the states, so the node is divided. The set of states that go in each division must be introduced, in groups that are indistinguishable. A number for a state can be introduced selecting firstly the node leaf where it was assigned, and then click on the corresponding state in the DF. Click on the state  $q_0$  of the DFA. The state number 0 will appear in the node leaf:



Add the rest of states to the nodes leaves, just as you have previously done with the state  $q_0$ , until arriving at the following image. To verify that the process is correct, click on **Check Node**.



Now we must continue with the rest of states. We will prove with the node leaf with states 0, 1, 4 and, 7, using the terminal  $a$ . Select **Set Terminal** and introduce  $a$ . There is a transition from  $q_0$  to  $q_5$  with  $a$ , which is in another node leaf, and the states  $q_1$ ,  $q_4$  and  $q_7$  have transitions among them, so they stay in the same node. Then, we already have the groups to be divided. To do this, select **Auto partition** and the states will be automatically inserted as shown in the following figure.



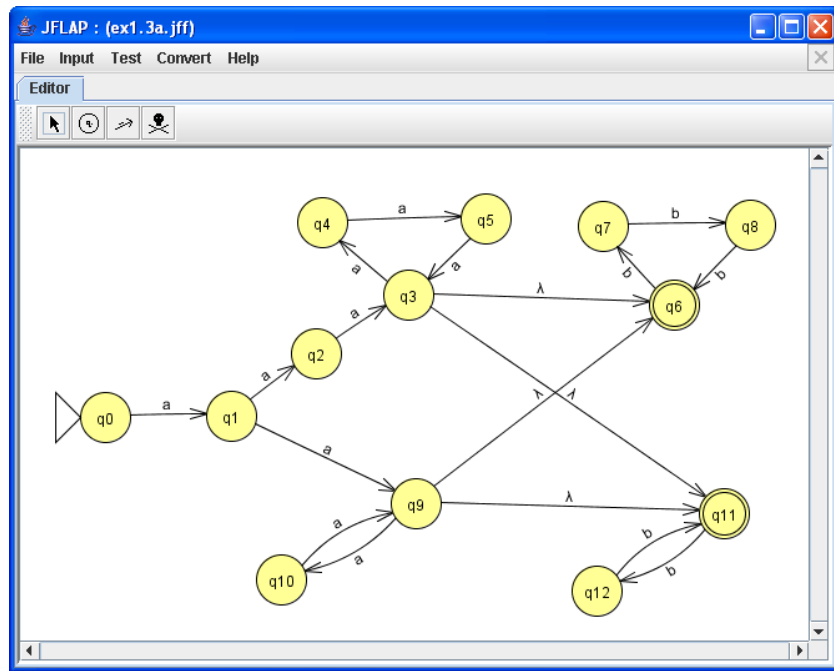
When the tree has been completed, the only visible option will be **Finish**. Select this button and the right part of the window will be replaced by the new states of the minimum DFA. There is a state for each node leaf of the tree. Now, add the rest of the arcs in the new DFA using the tool **Transition Creator**. In the original DFA there is one  $a$  from the state  $q_0$  to  $q_5$ . For this reason, the new DFA has a transition from the state  $q_1$  (that represents the old state  $q_0$ ), to  $q_2$  (representing the old  $q_5$ ).

Selecting **Hint**, JFLAP will automatically add transitions and selecting **Complete**, it will automatically complete the automata. Select **Done** to export the new AFD to a new window. The DFA with minimum states must be equivalent to the original DFA. Verify this using **Test: Compare Equivalence**.

### ***Building a 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.

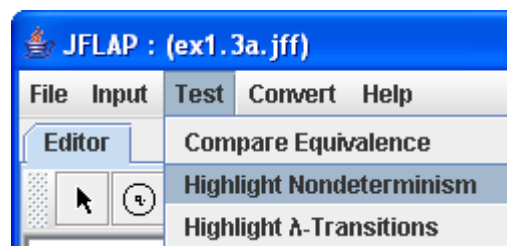




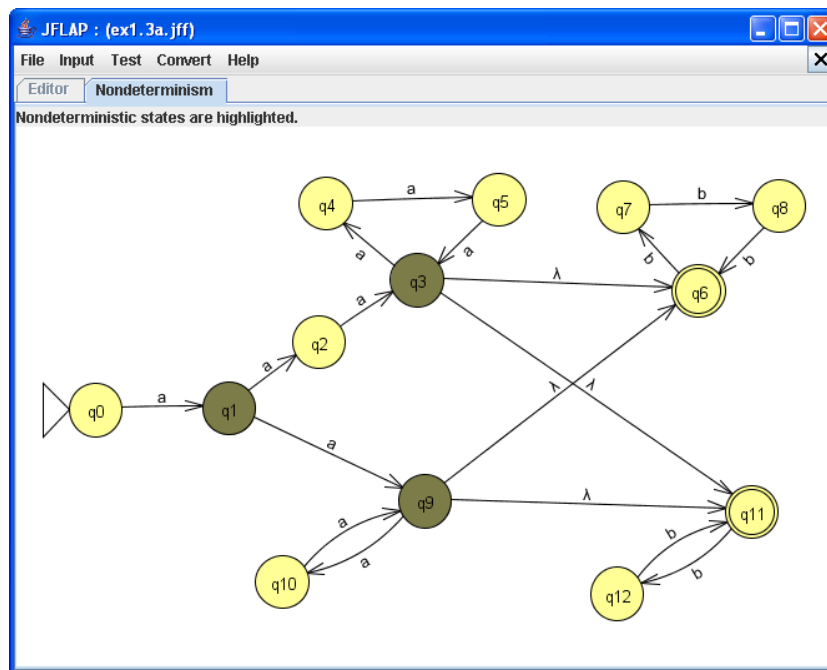
We can immediately tell that this is an NFA because of the four  $\lambda$ -transitions coming from  $q_3$  and  $q_9$ , but we might not be sure if we have spotted all the nondeterministic states. JFLAP can help with that.

### ***Highlighting Nondeterministic States***

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



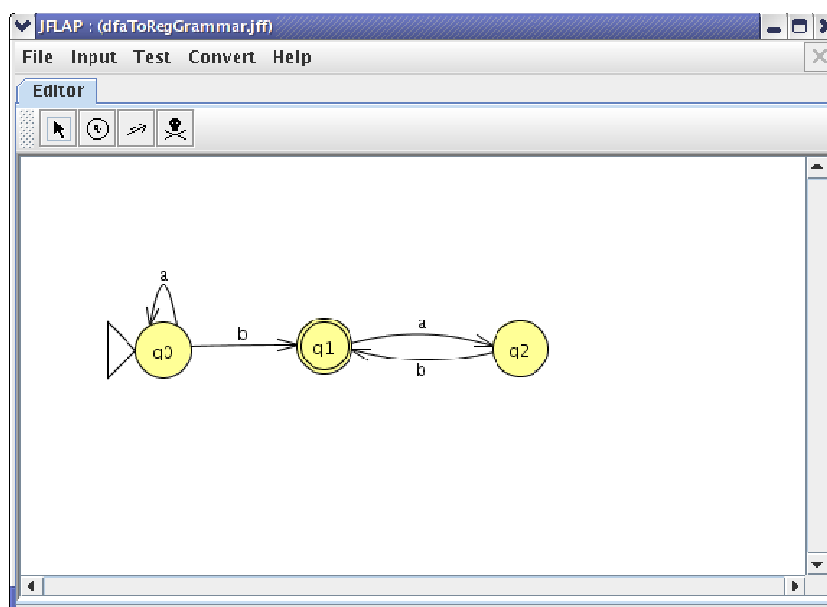
A new tab will appear with the nondeterministic states shaded a darker color:



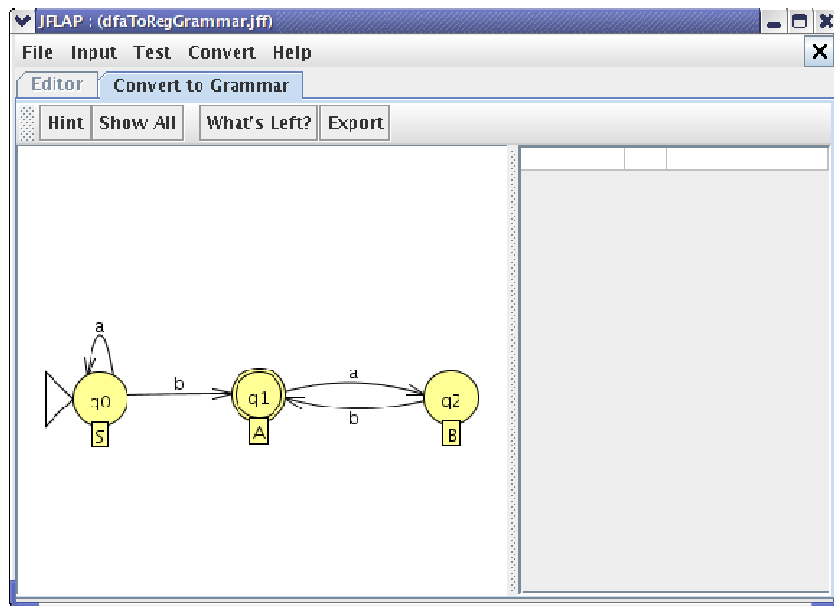
As we can see,  $q_3$  and  $q_9$  are indeed nondeterministic because of their outgoing  $\lambda$ -transitions. Note that they would both be nondeterministic even if they each had one  $\lambda$ -transition instead of two: only one  $\lambda$ -transition is needed to make a state nondeterministic. 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.

### ***Converting a DFA to a Regular Grammar***

This section specifically describes how one may transform one of these finite automata into a right-linear grammar. To get started, open JFLAP. Then, construct the nondeterministic finite automaton present in the screen below. When finished, your screen should look something like this:

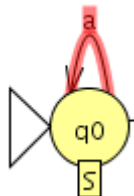


We will now convert this DFA into a regular grammar. Click on the “Convert → Convert to Grammar” menu option, and this screen should come up:



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

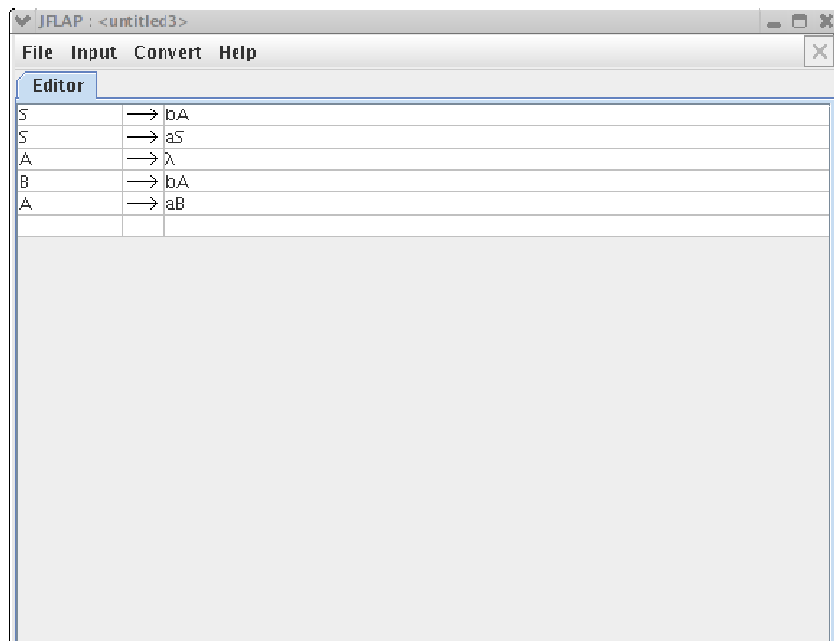


S	→	aS

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):

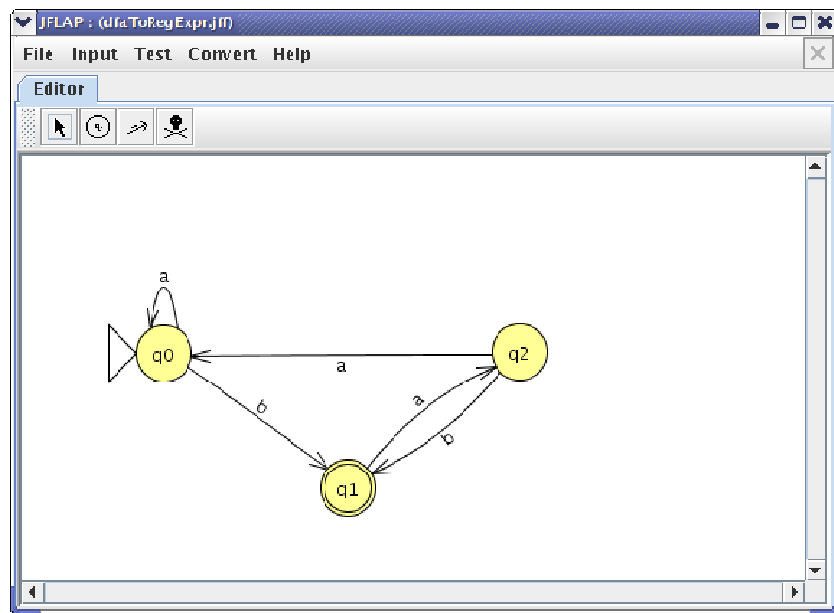
B	→	bA
S	→	bA
A	→	$\lambda$
A	→	aB
S	→	aS

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

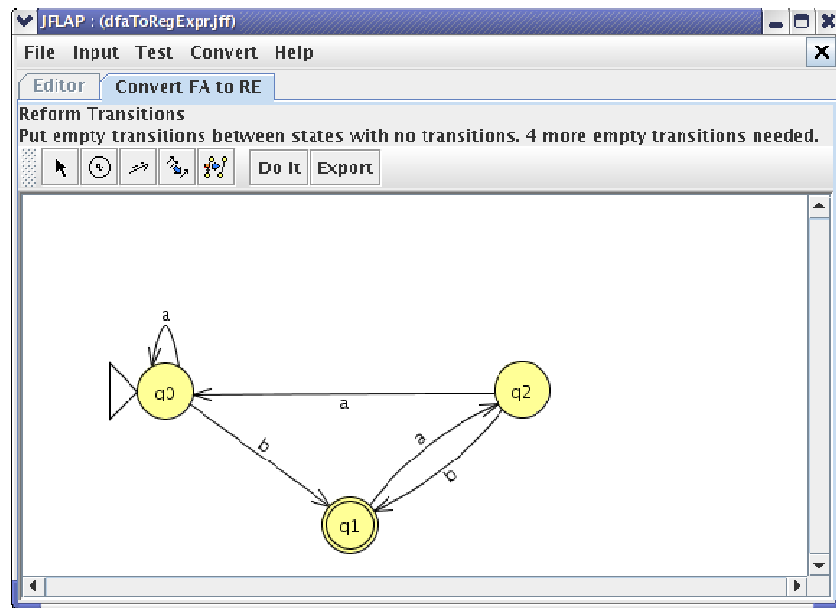


### ***Converting a FA to a Regular Expression***

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

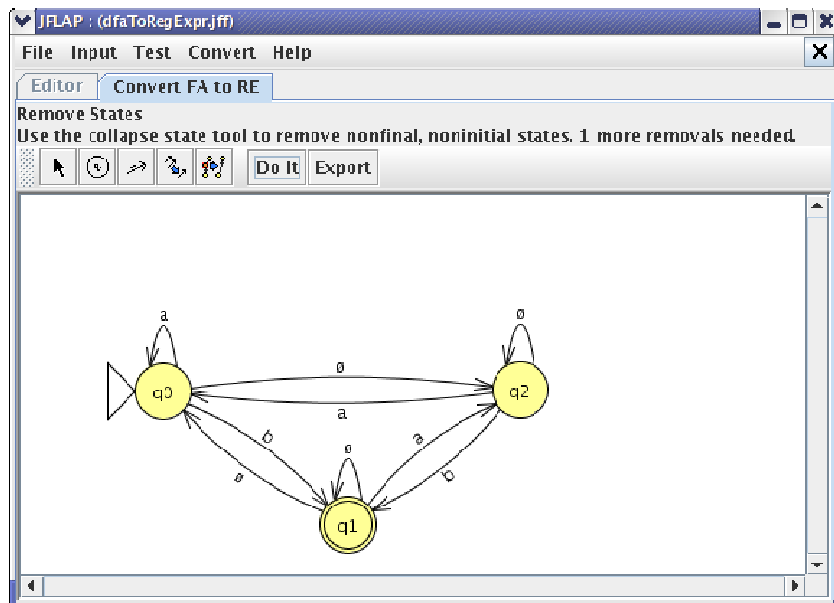


We will now convert this DFA into a regular expression. Click on the “Convert → Convert FA to RE” menu option, and this screen should come up:

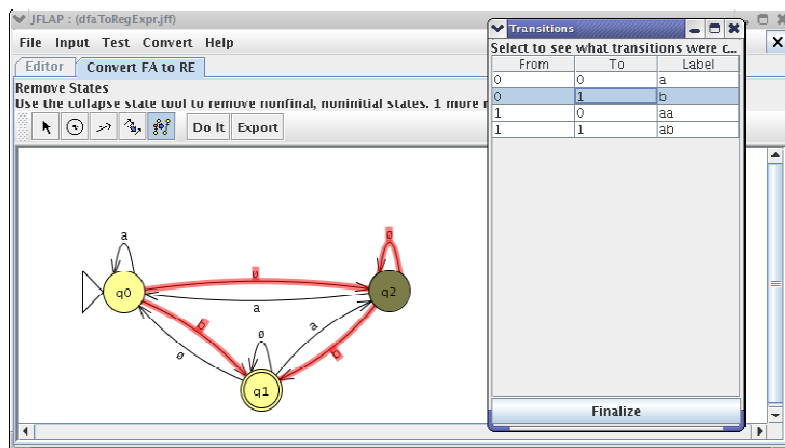


You can see that this is a fairly simple automaton, and that we are prompted for 4 new transitions. If this were a more complex automaton, with multiple final states, we would first have to create a single final state and then establish “ $\lambda$ ” transitions between the old final states and the new 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.

