

# JFLAP Startup

## Running JFLAP

Download JFLAP and the files referenced in this book from [www.jflap.org](http://www.jflap.org) to get started.

JFLAP is written in Java to allow it to run on a range of platforms. JFLAP requires that your computer have Java SE 1.4 or later installed. JFLAP works with Java 1.5. For the latest version, visit <http://java.sun.com/>. Mac OS X's latest Java release, if not already preinstalled, is available from <http://www.apple.com/java/>, or from Software Update.

With Java SE 1.4 or later installed on your computer, you may attempt to run JFLAP. JFLAP is distributed as an executable .jar (Java ARchive) file. JFLAP may be run as either an application or applet. The following table lists how to run the JFLAP.jar executable .jar as an application on your platform of choice.

Windows	Double click on JFLAP.jar; the file will appear as JFLAP if suffix hiding is on.
Unix & Linux	From a shell with the JFLAP.jar file in the current directory, enter the command <code>java -jar JFLAP.jar</code> .
Mac OS X	The Windows and Unix directions will work on a Mac.

## JFLAP Interface Primer

We cover universal elements of the JFLAP interface here. To begin, start JFLAP. When JFLAP has finished loading, a window will appear similar to that shown in Figure 1. This window offers a choice of major structures if you wish to create a new structure; alternatively, the **File** menu allows you to open an existing saved structure or quit JFLAP.

Throughout this book we shall review the creation of these structures. However, right now we are going to open a JFLAP saved file of an existing finite automaton (FA). From the **File** menu, choose **Open**. JFLAP allows users to save and open files that contain a single structure. Select and open the file `ex0.1a`. A new window will appear with an FA.

We refer to all the things you can do to a structure as *operators*. (It is not necessary to understand what the operators are doing at this point; our purpose is to describe JFLAP's interface.)

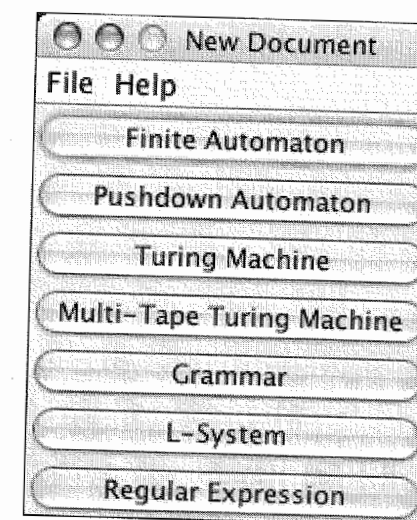


Figure 1: The window that appears when you start JFLAP.

Operators are typically activated through the menu items. Choose the menu item **Test : Highlight Nondeterminism**. (This activates an operator that shades nondeterministic states in an automaton, in this case  $q_0$  and  $q_1$ .) Next, choose the menu item **Test : Highlight  $\lambda$ -Transitions**. (This activates an operator that highlights  $\lambda$ -transitions in an automaton, in this case the arc labeled  $\lambda$ .) We chose these two operators because they require no intervention from the user.

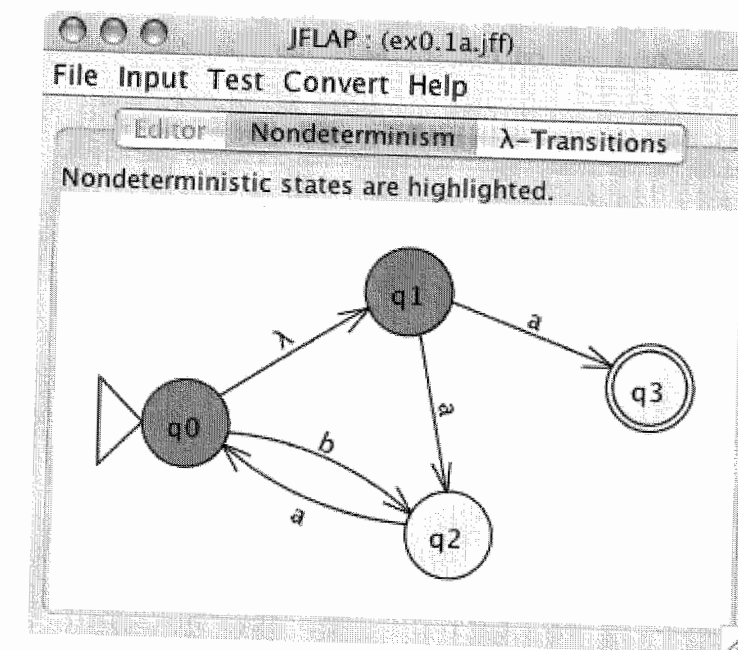


Figure 2: An illustration of the window for a structure, with three tabs active.

The window for a structure consists of a menu bar that contains operators you may apply to the structure, and a *tabbed interface* below the menu bar. Note that JFLAP *keeps everything related*

to a structure in a single window, and uses tabs to manage multiple operators active at the same time. The results of the two operators we invoked are displayed in tabs, so there are currently three tabs: **Editor**, **Nondeterminism**, and  **$\lambda$ -Transitions**. In Figure 2, the **Nondeterminism** tab is selected, indicating that the interface for the “highlight nondeterminism” operator is currently active and displayed. To switch to another operator, click on its tab. (Note that you cannot switch to the **Editor** tab at this time. This is because the other two currently active operators depend on the automaton not changing.)

We will now remove the **Nondeterminism** and  **$\lambda$ -Transitions** tabs. To get rid of a tab, select the menu item **File : Dismiss Tab**. This will remove the currently active tab. When it is gone, remove the other tab as well. (JFLAP prevents removal of the **Editor** tab.)

As a last step, peruse the contents of the **File** menu. Use **New** when you want to create a new structure; when **New** is selected, JFLAP will display the window shown in Figure 1 that allows you to choose a type of structure to create. The **Open**, **Save**, and **Save As** menu items allow you to read and write structures to files like any other application that deals with documents. The **Close** item will close the window of the structure. The **Print** item will print the currently active tab. **Quit** will stop JFLAP entirely.

## Chapter 1

# Finite Automata

A finite automaton is the first type of representation for a regular language that we will examine.

In this chapter we will construct a deterministic finite automaton (DFA) in JFLAP, illustrate several methods of simulating input on that automaton, discuss nondeterministic finite automata (NFAs) in JFLAP, and present simple analyses that JFLAP may apply to automata. We present a standard definition of a DFA in Sections 1.1–1.4, and show in the optional Section 1.5 how JFLAP handles a more general definition of a DFA with multiple character transitions.

### 1.1 A Simple Finite Automaton

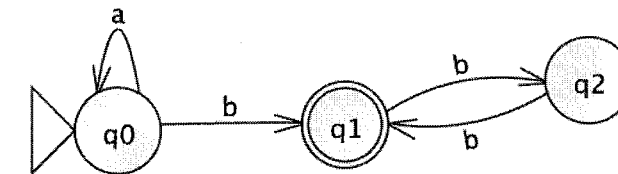



Figure 1.1: A finite automaton (FA), which recognizes the language of any number of  $a$ 's followed by any odd number of  $b$ 's.

In this section you will learn how to build automata in JFLAP by way of constructing, with help, the DFA that recognizes the language of strings of any number of  $a$ 's followed by any odd number of  $b$ 's (e.g.,  $ab$ ,  $bbb$ ,  $aabbbb$ ). This section will teach the essentials of automaton editing in JFLAP: creating and deleting states and transitions, moving existing states, editing existing transitions, and setting states to be initial and final. When you are done, you will have a machine like that pictured in Figure 1.1!


The first step is, of course, to start JFLAP. Once JFLAP is running, you begin building an FA by clicking on the button labeled **Finite Automaton**. A window will appear with (from top to bottom) a menu, a tab that says **Editor**, a tool bar, and a large blank area at the bottom.

### 1.1.1 Create States

All automata require a set of states. Before you can create states you must first activate the State Creator tool: click on the  button below the window's menu bar. This button will now appear shaded to indicate that tool is active.

The large blank area below the tools, called the *canvas*, is where the automaton is created and edited. Now that the State Creator tool is active, click on the canvas to create a state. A state will appear under the location where you clicked. As you will see, states in JFLAP are yellow circles with some identifying text inside. Click three more times in three other locations to create three more states. There will now be four states on the canvas, with the text  $q_0$ ,  $q_1$ ,  $q_2$ , and  $q_3$  to identify each of them.

### 1.1.2 Define the Initial State and the Final State


All automata require an initial state and a set of final states. In this automaton we will make  $q_0$  the initial state, and  $q_1$  the single final state. Select the Attribute Editor tool, by clicking the  button. Two of this tool's many functions are to define an initial state and to define the set of final states. (This tool's other functions are described in Section 1.1.5.)

Now that the Attribute Editor tool is selected, right-click on  $q_0$  (or, control-click if you are a Macintosh user with a single mouse button). A pop-up menu above the state will appear with several items, including two items **Final** and **Initial**. Select the item **Initial**. The state  $q_0$  will now have a white arrowhead appear to its left to indicate it is the initial state. Similarly, right-click on the state  $q_1$ , and select the item **Final**. The state  $q_1$  will now have a double outline instead of a single outline, indicating that this state is a member of the set of final states.

You may find it necessary to set a final state as nonfinal. To illustrate how, right-click on  $q_1$  once you have marked it as final. Notice that the item **Final** now has a check mark next to it. Select the item **Final** again. This will toggle  $q_1$  out of the set of final states. Before you proceed, you must of course put  $q_1$  in the set of final states again!

### 1.1.3 Creating Transitions

We will now create transitions. In this machine, three transitions are necessary: three on  $b$  from  $q_0$  to  $q_1$ ,  $q_1$  to  $q_2$ , and back again from  $q_2$  to  $q_1$ , and a loop transition on  $a$  for state  $q_0$ . We will create others for illustrating some special features, and for later illustration of the Deleter tool.

To create these transitions, select the Transition Creator tool, denoted by the  icon. The first transition we are going to create is the  $b$  transition from  $q_0$  to  $q_1$ . Once the Transition Creator tool is selected, press the mouse cursor down on the  $q_0$  state, drag the mouse cursor to the  $q_1$  state, and release the mouse button. A text field will appear between the two states. Type "b" and

press return. A new  $b$  transition from  $q_0$  to  $q_1$  will appear. By the same method, create the two  $b$  transitions from  $q_1$  and  $q_2$  and from  $q_2$  and  $q_1$ .


**Tip** || As an alternative to pressing return, you can stop editing a transition merely by doing something else like clicking in empty space (but not on a state!), or creating another transition by dragging between two other states. If you wish to cancel an edit of a transition, press Escape.

The next transition is  $q_0$ 's loop transition on  $a$ . Creating loop transitions on a state is just like other transitions: you press the mouse on the start state and release the mouse on the end state. However, because the start and end states are the same for a loop transition, this is the same as clicking on the state. So, click on state  $q_0$ , and enter "a" and press return, just as you did for the  $b$  transitions.

Lastly, create three transitions from  $q_0$  to  $q_3$ , the first on the terminal  $a$ , another on  $b$ , and a third on  $c$ . Notice that JFLAP stacks the transition labels atop each other.

**Tip** || If you are in the process of editing a transition from a state  $q_i$  to a state  $q_j$  and you wish to create another transition from state  $q_i$  to state  $q_j$  without having to use the mouse, press Shift-Return. This creates a new transition from  $q_i$  to  $q_j$  in addition to ending your editing of the current transition.



### 1.1.4 Deleting States and Transitions

You probably noticed that the automaton built requires three states, not four. This fourth state  $q_3$  and the transitions going to it are unnecessary and can be removed. Deleting objects has a tool all its own: click the  button to activate the Deleter tool.

First, we want to remove the transition on  $b$  from  $q_0$  to  $q_3$ . To delete this transition, click on the  $b$ . The  $b$  transition will be gone, leaving the  $a$  and  $c$  transitions. You can also click on the transition arrow itself to delete a transition: click on the arrow from  $q_0$  to  $q_3$ , and notice that the  $a$  transition disappears. The  $c$  transition remains. When you click on the arrow, the transition with the label closest to the arrow is deleted.

Deleting states is similar. Click on the state  $q_3$ . The state  $q_3$  will disappear, and notice that the  $c$  transition is gone as well. Deleting a state will also delete all transitions coming from or going to that state. You should now be left only with the other three states and the transitions between them.

### 1.1.5 Attribute Editor Tool

We already used the Attribute Editor tool  in Section 1.1.2, but it has many other functions related to modification of attributes of existing states and transitions. Select the Attribute Editor tool  once again as we walk through examples of its use.

### Setting states as initial or final

This tool may set states as initial or final states as described in Section 1.1.2.

### Moving states and transitions

When you initially placed the states for the FA built earlier you may not have arranged them in a logical order. To move a state, press on the state and drag it to a new location. Dragging a transition will likewise move its two associated states. Attempt this now by dragging states and transitions.

### Editing existing transitions

To edit an existing transition, simply click on it! Try clicking the transition from  $q_0$  to  $q_1$ . The same interface in which you initially defined this transition will appear on the transition and allow you to edit the input characters read by that transition.

### Labels

When you set the state  $q_0$  as the initial state and the state  $q_1$  as a final state, perhaps you noticed the menu item **Change Label**. Right-click on  $q_2$  and select **Change Label**. A dialog box will appear, asking for a label. When processing input, while the machine is in state  $q_2$ , we shall have processed an even number of  $b$ 's, so enter "even # of b's". A box will appear under the state with this label. By a similar token, label  $q_1$  "odd # of b's". To delete an existing label from a state choose the menu item **Clear Label** from the same menu. Alternatively, the menu item **Clear All Labels** will delete all labels from all states.

If you right-click in empty space, a different menu will appear, with the item **Display State Labels**. This will initially have a check mark next to it to indicate that it is active. Select it. The labels will become invisible. Hover the mouse cursor over  $q_2$ ; after a short time, a tool-tip will appear to display the label **even # of b's**. Right-click in empty space once more, and reactivate **Display State Labels**; the labels will appear again.

### Automatic layout

Right-click in empty space again. Notice the menu item **Layout Graph**. When selected, JFLAP will apply a graph layout algorithm to the automaton. While usually not useful for automata you produce yourself, many of JFLAP's algorithms automatically produce automata, often with large numbers of states. If you find JFLAP's first attempt at automatic layout inappropriate, this may alleviate the tedium of moving those states yourself.

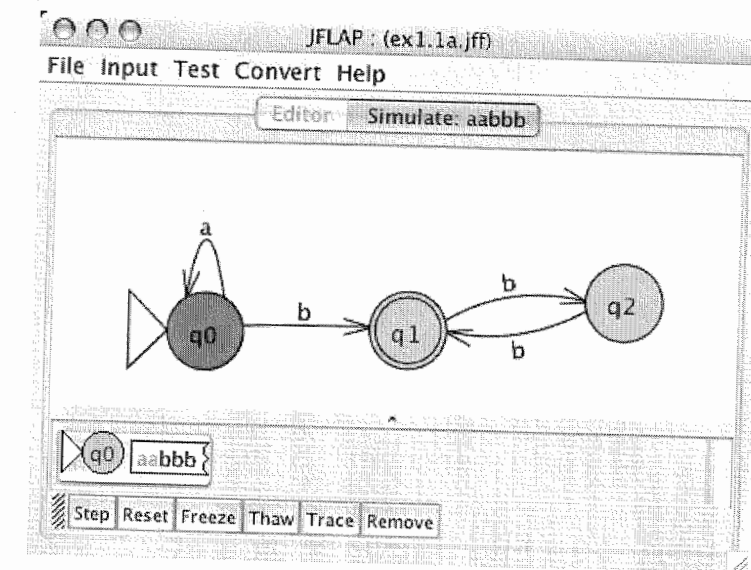



Figure 1.2: In the midst of the simulation of  $aabbb$  on our FA.

### Tip

In addition to activating a tool by clicking on its button in the toolbar, there are also shortcut keys available for quickly switching tools. For example, hover the mouse over the State Creator tool ; after a little while a tool-tip will appear with the text **(S)tate Creator**. The parentheses enclosing the **S** indicate that this is the shortcut key for the State Creator tool. Note that in spite of appearances, shortcut keys are really lower case, so do not press Shift when typing the shortcut key for a tool!

## 1.2 Simulation of Input

In this section we cover three of JFLAP's methods to simulate input on an automaton: stepping with closure, fast simulation, and multiple simulation. The fourth, stepping by state, is discussed briefly in Section 1.3.

### 1.2.1 Stepping Simulation

The stepping simulation option allows you to view every configuration generated by an automaton in its attempt to process an input string. Figure 1.2 shows a snapshot of a stepping simulation of the input string  $aabbb$  on the automaton you built in Section 1.1, also stored in file  $ex1.1a$ . The top portion of the window displays the automaton, with the state in the active configuration shaded darker. The portion below the automaton displays the current configuration. In Figure 1.2, notice the configuration is in state  $q_0$ , and that the first two characters  $aa$  are grayed-out, indicating that they have been read, while the three characters  $bbb$  are not grayed-out, indicating that they remain to be read.

### Try stepping

We shall walk through the process of stepping through input in an automaton. First, select the menu item **Input : Step with Closure**. A dialog box will ask for input for the machine: enter “aabb” and press Return or click **OK**.

Your window will now appear similar to Figure 1.2. The single configuration displayed will be on the initial state  $q_0$ , and have the unprocessed input *aabb*.

The tool bar at the bottom is your interface to the simulator. Click **Step**. The old configuration on  $q_0$  has been replaced with a new configuration, again on the state  $q_0$ , but with the character *a* read. Notice that the first character *a* in the input has been lightened a bit to indicate that it has been read. Click **Step** twice more, and it will go from  $q_0$  to  $q_0$  again, and then to  $q_1$ , with the input *bb* remaining.

Some of the operations in the tool bar below the configuration display act only on selected configurations. Click on the configuration; this will select it (or deselect it if it is already selected). A selected configuration is drawn shaded. Click **Remove**. Unfortunately, this deletes the only configuration! The simulator is useless. Oops! Click the **Reset** button; this will restart the simulation, so you can try again.

With the simulation back to its original state, click **Step** repeatedly (five times) until all the input is read. The configuration at this point should be drawn with a green background, indicating that it is an *accepting configuration*, and that the machine accepts the input. FA configurations are *accepting configurations* if all the input is read and it is in a final state. The configuration’s input is entirely gray, indicating that all the input has been read.

One can **Trace** a configuration to see its ancestry from the initial configuration. (Do not select a configuration; press **Trace** instead. An error message indicates that **Trace** requires a selected configuration!) Now select the single configuration, and click the **Trace** button. A window will show the ancestry of this configuration, starting with the initial configuration on top and the selected configuration on the bottom. When you’ve had a chance to look over the trace of the configuration, close this window.

To return to the editor, choose **File : Dismiss Tab** to dismiss the simulator.

### Failure

On the flip side of an accepting configuration is a *rejected configuration*. A rejected configuration is one which (1) does not lead to any more configurations and (2) is not accepting. Run a stepping simulation again, except this time with the input *aabb*. Since this has an even number of *b*’s the machine will not accept it. Click **Step** repeatedly, and note that eventually the configuration will turn red. This indicates that it is a rejected configuration.

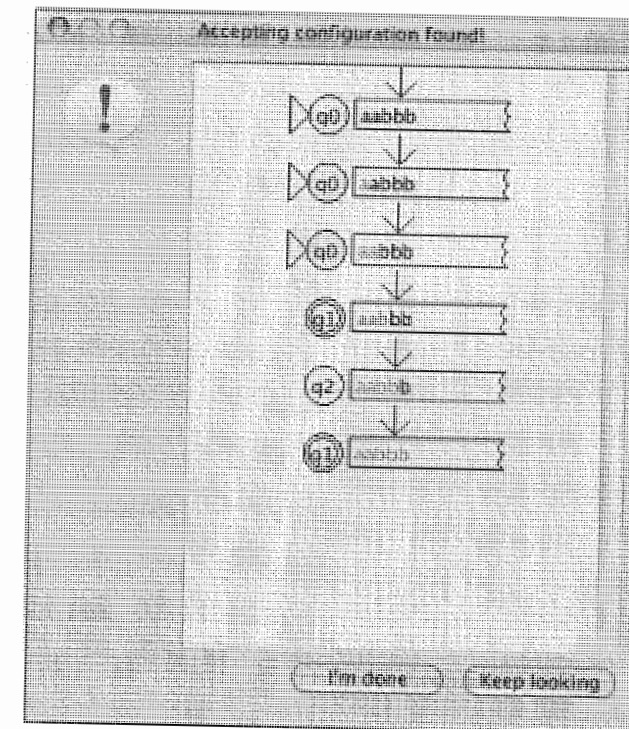


Figure 1.3: The result of performing a fast simulation of *aabb* on the automaton.

### 1.2.2 Fast Simulation

Stepping through simulation of input is fine, but **Fast Run** will reveal if the automaton accepts a string and, if it does, the series of configurations leading to that string’s acceptance without the bother of having to repeatedly step through the machine watching for accepting configurations.

Choose **Input : Fast Run**. When prompted for input, enter the same “aabb” string. The result after JFLAP determines that the machine accepts this input is shown in Figure 1.3. The trace of configurations, from top to bottom (i.e., from initial to accepting configuration), is displayed. Alternatively, if the machine did not accept this input, JFLAP would report that the string was not accepted.

Notice the two buttons near the bottom of the dialog box. **I’m Done** will close the display. **Keep Looking** is useful for nondeterministic machines and is covered later in Section 1.3.2.

### 1.2.3 Multiple Simulation

The third method for simulating input on an automaton is **Multiple Run**. This method allows one to perform multiple runs on a machine quickly. Select **Input : Multiple Run** now. (Your display will not resemble Figure 1.4 exactly, but do not worry!) The automaton is displayed to the left, and on the right is an empty table where you may enter inputs for multiple runs. One enters inputs in the **Input** column. Select the upper-left cell of this table, enter the input “aabb”, then

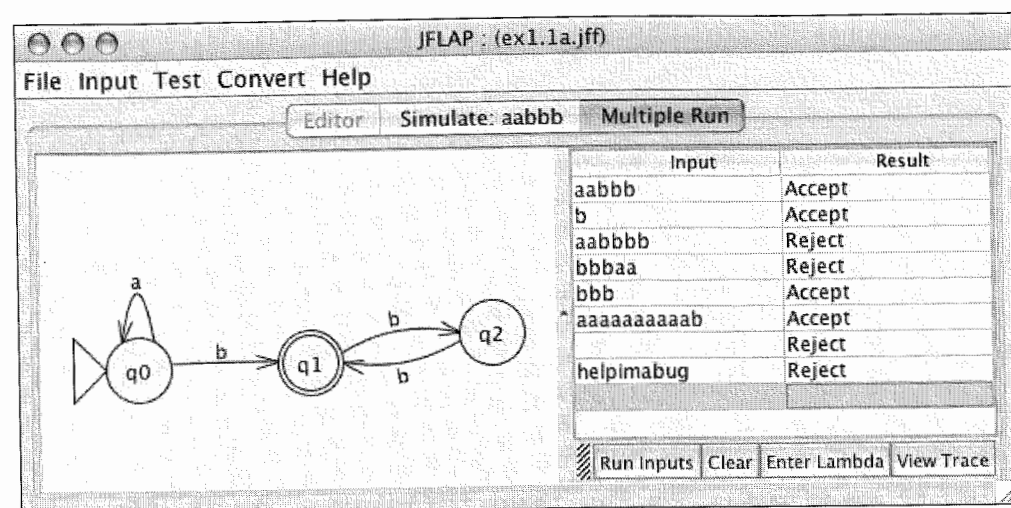


Figure 1.4: An example of simulating multiple entries. The second-to-last input is the empty string, entered with **Enter Lambda**.

press return. Notice that instead of one row there are now two: the table will grow to accommodate more entries as you enter them.

Continue entering various inputs you wish to test on the machine; whichever you choose is up to you. If you wish to make a lambda entry—that is, test to see if the automaton accepts the empty string—then *while entering an input*, click the **Enter Lambda** button near the bottom of the window, and that input field will hold the empty string. When you have entered all inputs and wish JFLAP to simulate all these strings, click **Run Inputs**. Notice that the **Result** column is now full of **Accept** and **Reject** entries, indicating whether an input was accepted or not. **View Trace** will show the trace of the last configuration generated for each selected run in the table. **Clear** will clear the table of all inputs.

**Tip** For convenience, the multiple run simulator will remember all inputs entered by the user between machines. For example, suppose you have one automaton, and perform multiple runs on that machine. If you later perform multiple run simulation on a different automaton those same inputs will appear.

### 1.3 Nondeterminism

In this section we will talk about NFAs in JFLAP, using the automaton pictured in Figure 1.5 as our example.

Either of two conditions imply that an FA is nondeterministic. The first condition is, if the FA has two transitions from the same state that read the same symbol, the FA is considered an NFA.

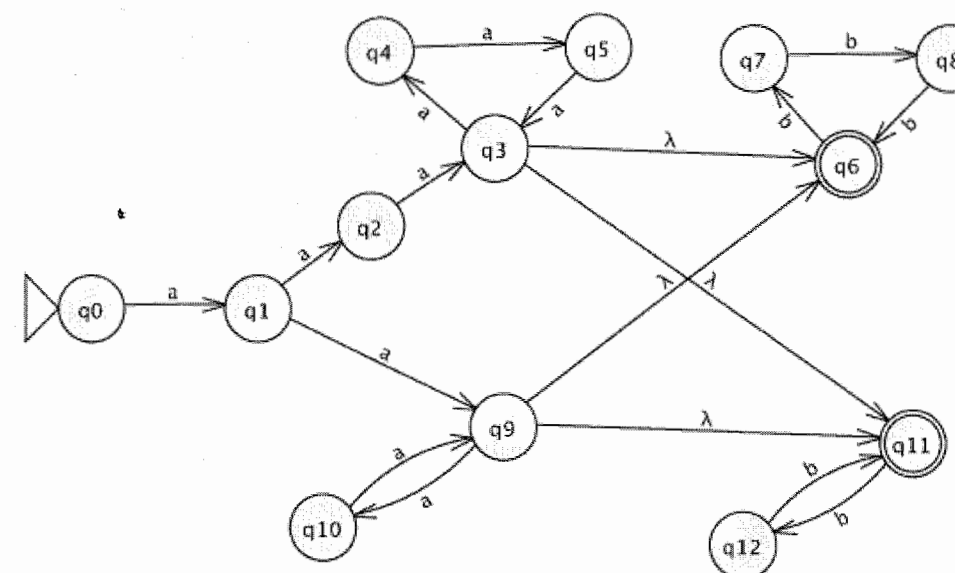


Figure 1.5: An NFA that accepts the language of a series of  $a$ 's followed by a series of  $b$ 's, where the number of  $a$ 's is nonzero and divisible by 2 or 3, and the number of  $b$ 's is divisible by 2 or 3.

For example,  $q_1$  of the FA in Figure 1.5 has two outgoing transitions on  $a$ . The second condition is: if the FA has any transitions that read the empty string for input, the FA is nondeterministic.

#### 1.3.1 Creating Nondeterministic Finite Automata

Creating an NFA is much the same as creating a DFA. Select **File : New**, and then select **Finite Automaton** to get a new window. In this window we will create the automaton shown in Figure 1.5, that accepts the language  $a^n b^m$ , where  $n > 0$  and is divisible by 2 or 3 and  $m \geq 0$  and is divisible by 2 or 3. The first step is to create the thirteen states of the automaton, and to make  $q_0$  the initial state and make  $q_6$  and  $q_{11}$  the final states.

Note that JFLAP numbers states in the order that you create them: the first state is  $q_0$ , the second  $q_1$ , and so on. It is important to respect this order: the following discussion assumes that you create the states in such an order that they are numbered as they are in Figure 1.5.

Notice the four transitions in Figure 1.5 with a  $\lambda$  (the Greek letter lambda). These  $\lambda$ -transitions are transitions on the empty string. To enter a  $\lambda$ -transition, create a transition, but leave the field empty. When you finish editing, a transition with the label  $\lambda$  will appear. Create the four  $\lambda$ -transitions from  $q_3$  and  $q_9$  to  $q_6$  and  $q_{11}$ .

Once you have created the  $\lambda$ -transitions, create the other transitions on the symbols  $a$  or  $b$ .

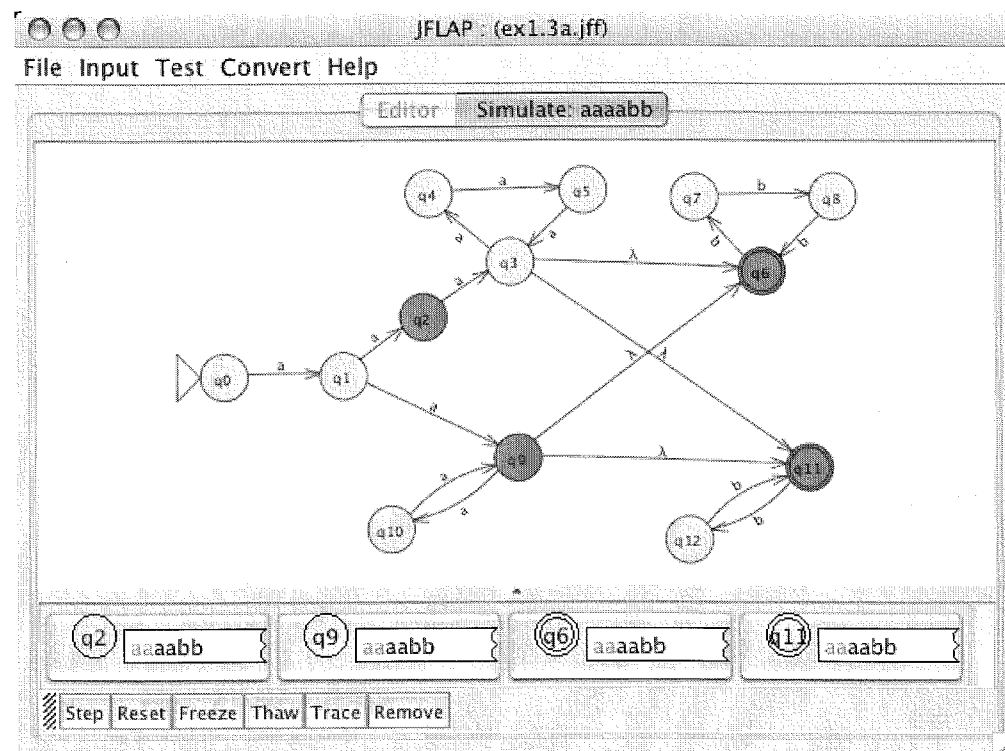


Figure 1.6: Step with closure simulation of  $aaaabb$  on our NFA after two steps.

### 1.3.2 Simulation

During simulation, input on a deterministic machine will produce a single path of configurations, while input on a nondeterministic machine may produce multiple paths of configurations. JFLAP's simulators have features to deal with this possibility.

#### Stepping simulation: Step with Closure

Select the menu item **Input : Step with Closure** and input the string "aaaabb", that is, four  $a$ 's followed by two  $b$ 's. This is a string that will eventually be accepted since the number of  $a$ 's is nonzero and divisible by 2 and the number of  $b$ 's is divisible by 2. After you enter this input, you should see the familiar step simulator, with a starting configuration on  $q_0$  with all the input remaining to be processed. Click **Step** once to move this configuration to  $q_1$ . However, if you click **Step** a second time you will see a rather unfamiliar sight, as shown in Figure 1.6.

Notice that there are four configurations in your simulator. This is because your machine is nondeterministic: The last configuration was on  $q_1$  with the unread input  $aaabb$ , and  $q_1$  has  $a$  transitions to  $q_2$  and  $q_9$ . However, what two configurations on  $q_6$  and  $q_{11}$ ? These configurations are due to the  $\lambda$ -transitions. When a configuration proceeds to a state  $q_i$ , **Step with Closure** creates configurations not only for  $q_i$ , but for all states reachable on  $\lambda$ -transitions from  $q_i$ . The set

of states reachable from  $q_i$  on  $\lambda$ -transitions is called the *closure* of  $q_i$ . So, when the configuration in  $q_9$  with the remaining input  $aabb$  was created, configurations for  $q_6$  and  $q_{11}$  were created as well because the closure of  $q_9$  includes  $q_6$  and  $q_{11}$ .

As you may have figured out, of these two paths of configurations, the only one that will eventually lead to an accepting configuration is the configuration on  $q_9$ . Click on this configuration to select it. With the configuration selected, click **Freeze**. The configuration will appear tinted light blue! Now try stepping again: While the other configurations move on (and are rejected), that configuration will not progress! Frozen configurations do not progress when the simulator steps. With that configuration still selected, click **Thaw**. **Thaw** "unfreezes" selected configurations. Click the **Step** button once more, and the now unfrozen configuration will continue, and one of its nondeterministic paths will be accepted.

Select the accepting configuration and click **Trace** to view the series of configurations that led to the accepting configuration. Notice that there is a configuration from  $q_{10}$  directly to  $q_{11}$ , even though there is no transition from  $q_{10}$  to  $q_{11}$ . In stepping by closure one does not explicitly traverse  $\lambda$ -transitions in the same sense that one traverses regular transitions: Instead, no configuration was ever generated for  $q_9$ , and the simulator implicitly traversed the  $\lambda$ -transition.

When you have finished, dismiss the simulator tab.

#### Stepping simulation: Step by State

Select the menu item **Input : Step by State**, and input the string "aaaabb". In stepping by state, the closure is not taken, so the simulator explicitly traverses  $\lambda$ -transitions. If you step twice, you will have configurations in  $q_2$  and  $q_9$ , but not the configurations in  $q_6$  and  $q_{11}$  that we saw when stepping by closure.

Notice that the unread input on the  $q_9$  configuration is  $aabb$ . If you step again, the configuration on  $q_9$  will split into three configurations, two of which are on  $q_6$  and  $q_{11}$ . The  $\lambda$ -transition was taken explicitly over a step action. If you continue stepping until an accepting configuration is encountered and run a trace, the configuration after  $q_{10}$  is on  $q_9$ , which then proceeds to  $q_{11}$  after explicitly taking the  $\lambda$ -transition.

Though stepping by state is in some ways less confusing, stepping with closure is often preferred because it guarantees that each step will read an input symbol.

#### Fast simulation

The fast simulator has some additional features specifically for nondeterminism. Select **Input : Fast Run**, and enter the string "aaaaaabb". Once you enter this, JFLAP will display one trace of accepting configurations.

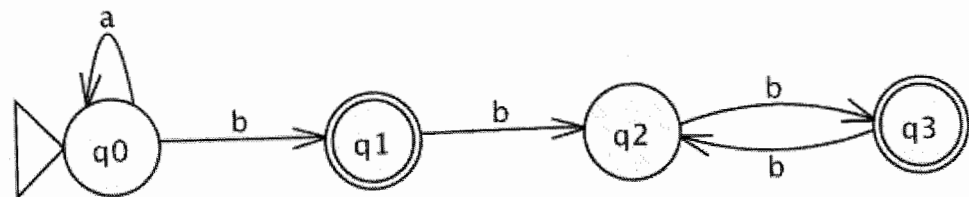


Figure 1.7: Another FA, which also recognizes the language of the automaton in Figure 1.1.

The button **Keep Looking** is useful for nondeterministic machines, where multiple branches of configurations may accept the same input. Note that there are six  $a$ 's. Since six is divisible by both two and three, there will be two paths of configurations that accept this input: one path leads through state  $q_3$  (which verifies that the number of  $a$ 's is divisible by three), and another path leads through state  $q_9$  (which verifies that the number of  $a$ 's is divisible by two). The trace through either  $q_3$  or  $q_9$  should be visible now. Click **Keep Looking**, and it will search for and display the trace through the other state. Click **Keep Looking** again. JFLAP will display a message, **2 configurations accepted, and all other possibilities are exhausted**, which indicates that no other accepting configurations are possible.

### Multiple simulation

Nondeterministic machines may produce multiple configuration paths per run. However, the multiple run simulator's ability to view traces of selected runs will present only a single trace for each run. Specifically, this feature displays only the trace of the *last* configuration generated for a run. This means that for an accepting run JFLAP displays the trace of the first accepting configuration encountered, and further for a rejecting run displays the trace of the last configuration rejected, which may not provide enough information. Viewing a run in the stepwise simulator can give a more complete picture if you want to debug a nondeterministic machine.

## 1.4 Simple Analysis Operators

In addition to the simulation of input, JFLAP offers a few simple operators from the **Test** menu to determine various properties of the automaton.

### 1.4.1 Compare Equivalence

This operator compares two finite automata to see if they accept the same language. To illustrate how this works, we shall load an automaton that recognizes the same language as the automaton we have abused throughout much of this chapter: the automaton shown in Figure 1.7, stored in file **ex1.4a**. Open this file. Also, open the file **ex1.1a**; this contains the automaton of Figure 1.1.

You will now have two windows, one with the original automaton of Figure 1.1 (presumably titled **ex1.1a**), the other with the automaton of Figure 1.7 (presumably titled **ex1.4a**). Choose the menu item **Test : Compare Equivalence** from the **ex1.4a** window. A prompt will appear where you may choose from the names of one other automaton (i.e., the title of another automaton's window) from a list. After you select the original automaton's window's name (again, presumably **ex1.1a**), click **OK**. You will then receive a dialog box telling you that they are equivalent! Dismiss this dialog. Edit the Figure 1.7 automaton so that the  $b$  transition from  $q_0$  to  $q_1$  is instead an  $a$  transition (so that the automaton now recognizes strings with any nonzero number of  $a$ 's and an even number of  $b$ 's), or make whatever other change is to your liking so that the automaton no longer recognizes the same language as the original. Repeat the test for equivalence, and this time you will receive a notice that it does not accept the same language.

Close the two files, but do not save the changes from the modified **ex1.4a**.

### 1.4.2 Highlight Nondeterminism

This operator will show the user which states in an automaton are nondeterministic states. Consider again the automaton in Figure 1.5, stored in the file **ex1.3a**. Load this file. The state  $q_1$  is obviously nondeterministic, and JFLAP considers all states with outgoing  $\lambda$ -transitions to be nondeterministic states, so  $q_3$  and  $q_9$  are nondeterministic. Select **Test : Highlight Nondeterminism**: a new view will display the automaton with these states highlighted.

### 1.4.3 Highlight $\lambda$ -Transitions

This operator will highlight all  $\lambda$ -transitions. Here we use the same automaton we built in Section 1.3.1, the automaton shown in Figure 1.5 and stored in the file **ex1.3a**. Load this file if it is not already present. When you select **Test : Highlight  $\lambda$ -Transitions**, a new view will display the automaton with the four  $\lambda$ -transitions highlighted.

## 1.5 Alternative Multiple Character Transitions\*

JFLAP provides a more general definition of an FA, allowing multiple characters on a transition. This can result in simpler FAs. Pictured in Figure 1.8 is a five-state NFA that accepts the same language as the thirteen-state NFA in Figure 1.5. Notice that the six transitions that are not  $\lambda$ -transitions are on multiple symbols, for example,  $aaa$  from  $q_0$  to  $q_1$ . A configuration may proceed on an  $n$  character transition of  $s_1s_2\dots s_n$  if the next unread input symbols are  $s_1$ ,  $s_2$ , and so on through  $s_n$ .

We will now run a simulation on this NFA. Load the file **ex1.5a**, select **Step With Closure**, and enter the same **aaaabb** string we used in Section 1.3.2. After you enter the input, you will see



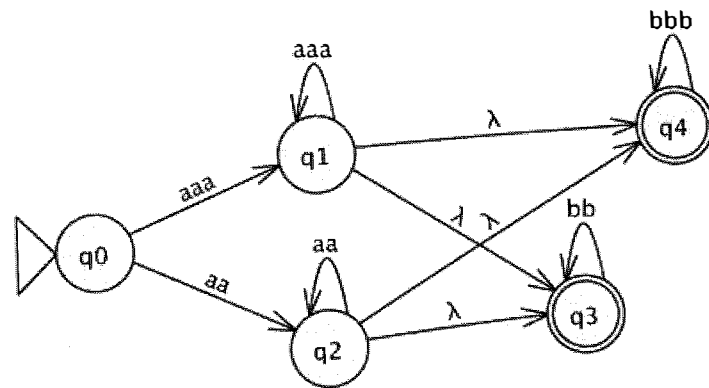


Figure 1.8: An NFA equivalent to that of Figure 1.5.

the familiar step simulator, with a starting configuration on  $q_0$  with all the input remaining to be processed. Click **Step** once and you will see six configurations! There are two configurations for  $q_3$ , one closure from  $q_1$  and one closure from  $q_2$ . Note that these two configurations have different amounts of remaining input since the transitions to  $q_1$  and  $q_2$  process a different amount of input. Similarly, there are two configurations for  $q_4$ . Stepping twice more results in acceptance in  $q_3$ .

By allowing multiple character transitions, the first condition for FA nondeterminism in Section 1.3 changes. The first condition is now the following: if the FA has two transitions from the same state that read strings  $A$  and  $B$ , where  $A$  is a prefix of  $B$ , the FA is considered an NFA. For example, note that  $q_0$  is a nondeterministic state: it has two transitions, one from  $aaa$  and the other from  $aa$ ;  $aa$  is a prefix of  $aaa$ , so the FA is nondeterministic. The NFA would use both of these transitions while simulating the string  $aaaabb$ .

## 1.6 Definition of FA in JFLAP

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

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

$\Sigma$  is the finite input alphabet

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

$q_s \in Q$  is the initial state

$F \subseteq Q$  is the set of final states

Users reading only Sections 1.1–1.4 will want to use a simpler definition of  $\delta$ . In that case, for a DFA  $\delta$  is the transition function  $\delta : Q \times \Sigma \rightarrow Q$ , and for an NFA  $\delta$  is the transition function  $\delta : Q \times \Sigma \cup \{\lambda\} \rightarrow 2^Q$ .

For those users reading Section 1.5, note that JFLAP allows for multiple characters on a transition. These multiple character transitions complicate the definition of the transition function's domain: the set  $Q \times \Sigma^*$  is of infinite cardinality, though the transition function requires a finite domain.  $\Sigma^*$  means a string of 0 or more symbols from the input alphabet.

## 1.7 Summary

In Section 1.1 you learned how to create a deterministic finite automaton (DFA) in JFLAP. The editor for an automaton has a tool bar along the top portion of the window, and the automaton display on the bottom portion of the window. You create states with the  $\odot$  tool, create transitions with the  $\rightarrow$  tool, delete states and transitions with the  $\times$  tool, and edit attributes (position, labels, setting final and initial) of existing states and transitions with the  $\text{↖}$  tool.

In Section 1.2 you learned how to simulate input on automata. Each simulator accepts an input string and determines if the automaton accepts that input. The step simulator is useful if you are interested in seeing every configuration generated by a machine as it attempts to read your input. The fast simulator is useful if you are interested only in those configurations that led to an accepting configuration. The multiple input simulator is useful if you are interested in running many inputs on an automaton quickly.

In Section 1.3 you learned about creating and simulating input on a nondeterministic finite automaton (NFA). Leaving the field blank when creating a transition will produce a  $\lambda$ -transition. While simulating input, the step simulator may display multiple configurations at once as the machine follows different paths attempting to read the input. The fast simulator can search for multiple branches of nondeterminism accepting the same input.

In Section 1.4 we presented three analysis operators available from the **Test** menu. **Compare Equivalence** checks if two finite automata accept the same language. **Highlight Nondeterminism** highlights nondeterministic states, and **Highlight  $\lambda$ -Transitions** highlights  $\lambda$ -transitions.

In Section 1.5 we presented an alternative definition of an FA that allows for multiple characters on a transition. This can lead to an FA with a smaller number of states.

In Section 1.6 we presented JFLAP's formal definition of a finite automaton, which corresponds to Section 1.5. We also presented a simpler definition corresponding to Sections 1.1–1.4.

## 1.8 Exercises

1. Build FAs with JFLAP that accept the following languages:

- The language over  $\Sigma = \{a\}$  of any odd number of  $a$ 's.
- The language over  $\Sigma = \{a\}$  of any even number of  $a$ 's.
- The language over  $\Sigma = \{a, b\}$  of any even number of  $a$ 's and any odd number of  $b$ 's.
- The language over  $\Sigma = \{a, b\}$  of any even number of  $a$ 's and at least three  $b$ 's.

- (b) Repeat part (a), but with the language of strings  $a_n b_n c_n \dots a_1 b_1 c_1 a_0 b_0 c_0$ .
- (c) Repeat part (a), but with the language of strings  $a_0 \dots a_n b_0 \dots b_n c_0 \dots c_n$ .
8. Given two FAs  $A$  with language  $L_A$  and  $B$  with language  $L_B$ , you can use JFLAP's **Compare Equivalence** operator to determine whether or not  $L_A = L_B$ . Can you devise a general method using JFLAP to determine whether  $L_A \subseteq L_B$  (i.e.,  $B$  accepts every string  $A$  accepts) using **Compare Equivalence**? (Yes, part of your instructions may, indeed must, involve editing  $A$  or  $B$ . Your method must produce the right answer for any two FAs!)

## Chapter 2

# NFA to DFA to Minimal DFA

This chapter shows how each NFA can be converted into an equivalent DFA, and how each DFA can be reduced to a DFA with a minimum number of states. Although an NFA might be easier to construct than a DFA, the NFA is usually not efficient to run, as an input string may follow several paths. Converting an NFA into an equivalent DFA ensures that each input string follows only one path. The NFA to DFA algorithm in JFLAP combines similar states reached in the NFA into one state in the DFA. The DFA to minimum state DFA algorithm in JFLAP determines which states in the DFA have similar behavior with respect to incoming and outgoing transitions and combines these states, resulting in a minimal state DFA.

### 2.1 NFA to DFA

In this section we use JFLAP to show how to convert an NFA into an equivalent DFA. The idea in the conversion is to create states in the DFA that represent multiple states in the NFA. The start state in the DFA represents the start state in the NFA and any states reachable from it on  $\lambda$ . For each new state in the DFA and each letter of the alphabet, one determines all the reachable states from the corresponding NFA states and combines them into a new state for the DFA. This state in the DFA will have a label that will contain the state numbers from the NFA that would be reachable in taking the same path.

#### 2.1.1 Idea for the Conversion

Load the NFA in file `ex2.1a` as shown in Figure 2.1. We will refer to this example in explaining the steps in converting this NFA to a DFA.

First examine the choices that occur when the NFA processes input. Select **Input : Step with Closure** and enter the input string "aabbbaa" and press return. Clicking **Step** once shows that processing  $a$  can result in arriving in both states  $q_0$  and  $q_1$ . Clicking **Step** six more times shows

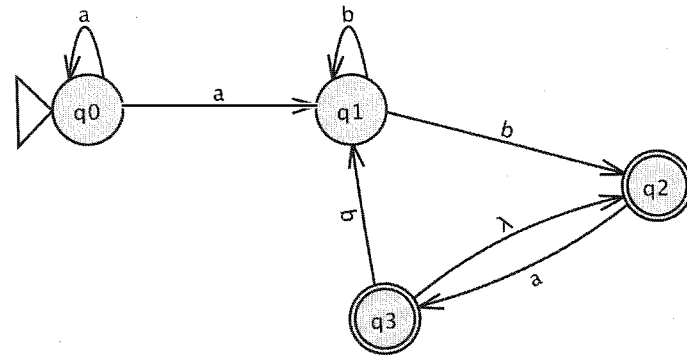


Figure 2.1: Example from file ex2.1a.

that there are always three configurations (one of which is rejected), and results in two paths of acceptance in states  $q_2$  and  $q_3$ .


The states in the constructed DFA will represent combined states from the NFA. For example, processing an  $a$  resulted in either state  $q_0$  or  $q_1$ . The DFA would have a state that represents both of these NFA states. Processing  $aabbaa$  resulted in reaching final states  $q_2$  and  $q_3$ . The DFA would have a state that represented both of these NFA states. Dismiss the tab for the step run (select **File : Dismiss Tab**) to go back to the NFA editor.

### 2.1.2 Conversion Example

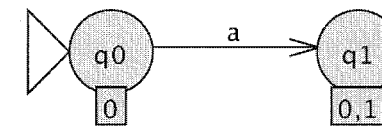
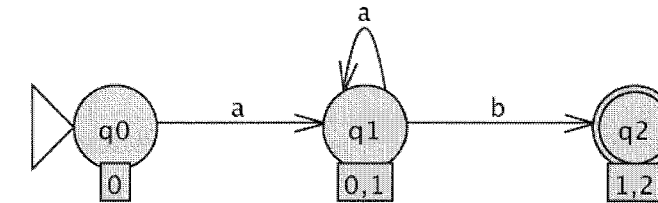
Now we will convert the NFA to a DFA (select **Convert : Convert to DFA**), showing the NFA on the left and the first state of the DFA on the right. The initial state in the DFA is named  $q_0$  and has the label 0, meaning it represents the  $q_0$  state from the NFA.

#### Tip

The NFA may be tiny. Adjust the size in one of two ways: either resize the window, or drag the vertical bar between the NFA and the DFA to the right. In addition, the states in the DFA can be dragged closer to each other, resulting in larger states.

We will now add the state that is reachable from  $q_0$  on the substring  $a$ . Select the Expand Group on Terminal tool . Click and hold the mouse on state  $q_0$ , drag the cursor to where you want the next state, and release it. When prompted by **Expand on what terminal?**, enter “a” and press return. When prompted by **Which group of NFA states will that go to on a?**, enter the numbers of the states that are reachable from  $q_0$  on an  $a$ . In this case enter “0,1”. (These NFA states could also be entered with a blank separator and with or without the  $q$ , such as “q0,q1”.) The new state  $q_1$  appears in Figure 2.2.

Use the Attribute Editor tool you learned about in Chapter 1 to move states around if you don't like their placement.

Figure 2.2: Expansion of state  $q_0$  on  $a$ .Figure 2.3: Expansion of  $a$  and  $b$  from state  $q_1$ .

Try expanding DFA state  $q_0$  on the terminal  $b$ . Since there are no paths from NFA state  $q_0$  on a  $b$ , a warning message is displayed.


Next expand the DFA state  $q_1$  on the terminal  $a$ . Note that DFA state  $q_1$  represents both states  $q_0$  and  $q_1$  from the NFA. In the NFA, state  $q_0$  on an  $a$  reaches states  $q_0$  and  $q_1$ , and state  $q_1$  on an  $a$  reaches no state. The union of these results (0, 1) are the states reachable by DFA state  $q_1$ , which happens to be the DFA state  $q_1$ . Upon the completion of the expansion a transition loop labeled  $a$  is added to DFA state  $q_1$ . Now expand DFA state  $q_1$  on  $b$ . The result of these two expansions is shown in Figure 2.3. Why is DFA state  $q_2$  a final state? If a DFA state represents any NFA state that is a final state, then the substring processed is accepted on some path, and thus the DFA state also must be a final state. NFA state  $q_2$  is a final state, so DFA state  $q_2$  (representing NFA states  $q_1$  and  $q_2$ ) is a final state.

Expand DFA state  $q_2$  on  $a$ . This state is represented by NFA states  $q_1$  and  $q_2$ . NFA state  $q_1$  does not have an  $a$  transition. NFA state  $q_2$  on an  $a$  reaches state  $q_3$  and due to the  $\lambda$ -transition also reaches state  $q_2$ .

#### Note

In using the Expand Group Terminal tool, if the destination state already exists, then drag to the existing state and you will be prompted only for the terminal to expand. Thus, to add a loop transition, just click on the state.

Expand DFA state  $q_2$  on  $b$  by clicking on state  $q_2$ . You are prompted for the  $b$ , but not the states reachable, as that is interpreted as your selected state (itself in this case). The resulting DFA is shown in Figure 2.4.

There is another way to expand a state—the State Expander tool . When one selects this tool and clicks on a state, all arcs out of the state are automatically expanded. In Figure 2.5 state  $q_3$  was selected and expanded on both  $a$  and  $b$ , resulting in a new state  $q_4$ .

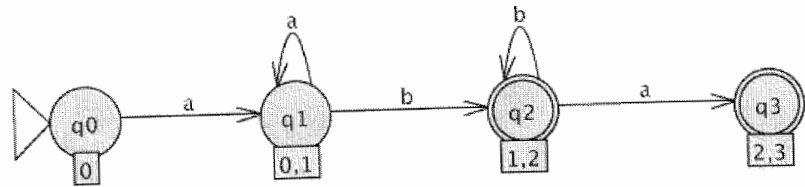
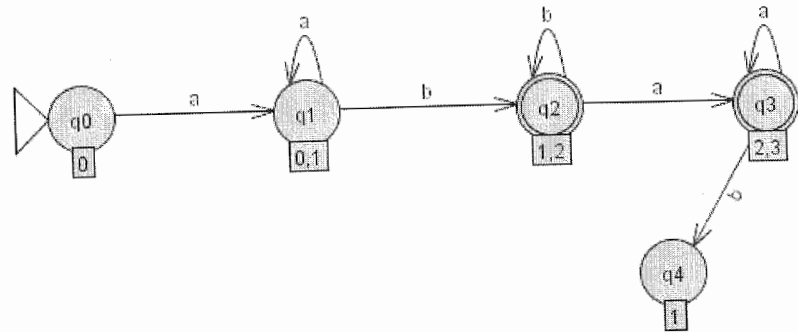
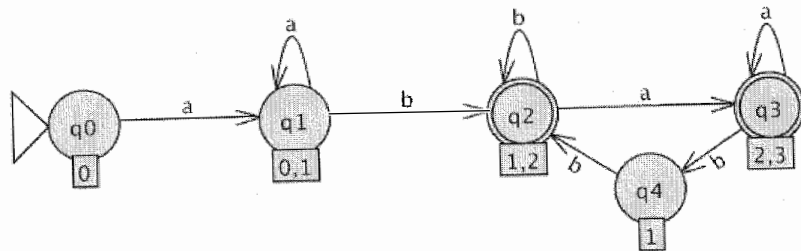
Figure 2.4: Expansion of  $a$  and  $b$  from state  $q_2$ .Figure 2.5: State Expander tool applied to state  $q_3$ .

Figure 2.6: The completed DFA.

Is the DFA complete? Select the **Done?** button. If the DFA is not complete, a message indicating items missing is displayed. At this time, one transition is missing.

Expand DFA state  $q_4$  on  $b$  by going back to the Expand Group on Terminal tool. Note that  $q_4$  on  $b$  goes to the existing DFA state  $q_2$ . Click on state  $q_4$ , drag to state  $q_2$ , and release. You will be prompted for the terminal only.

Is the DFA complete? Select the **Done?** button. The DFA is complete and is exported to a new window. The complete DFA is shown in Figure 2.6. Alternatively, the **Complete** button can be selected at any time during the construction process and the complete DFA will be shown.

The constructed DFA should be equivalent to the NFA. To test this, in the DFA window select **Test : Compare Equivalence**. Select file `ex2.1a`, the name of the NFA, and then press return. The two machines are equivalent.

### 2.1.3 Algorithm to Convert NFA $M$ to DFA $M'$

We describe the algorithm to convert an NFA  $M$  to a DFA  $M'$ . We first define the *closure* of a set of states to be those states unioned with all states reachable from these states on a  $\lambda$ -transition.

1. The initial state in  $M'$  is the closure of the initial state from  $M$ .
2. For each state  $q'$  in  $M'$  and each terminal  $x$  do the following:
  - (a) States  $q$  and  $r$  are states in  $M$ . For each state  $q$  that is in state  $q'$ , if  $q$  on an  $x$  reaches state  $r$  on an  $x$ , then place state  $r$  in new state  $p'$ .
  - (b)  $p' = \text{closure}(p')$
  - (c) If another state is equivalent to state  $p'$  (represents the same states from  $M$ ), then set  $p'$  to the state already existing.
  - (d) Add the transition to  $M'$ :  $q'$  to  $p'$  on an  $x$ .
3. Each state  $q'$  in  $M'$  is a final state if it contains a final state from  $M$ .

## 2.2 DFA to Minimal DFA

In this section we show how to convert a DFA to a minimal state DFA. Consider two states  $p$  and  $q$  from a DFA, each processing a string starting from their state. If there is at least one string  $w$  such that states  $p$  and  $q$  process this string and one accepts  $w$  and one rejects  $w$ , then these states are *distinguishable* and cannot be combined. Otherwise, states  $p$  and  $q$  “act the same way,” meaning that they are *indistinguishable* and can be combined.

### 2.2.1 Idea for the Conversion

Load the DFA in Figure 2.7 (file `ex2.2a`). We will refer to this example to explain the steps to convert this DFA to a minimal state DFA.

We would like to examine pairs of states to see if they are distinguishable or not. To do this we will need two separate windows for this DFA. JFLAP lets you open only one copy of each file, so if you try to open the same file again, JFLAP will show just the one window. Instead we will make a duplicate copy of this file by saving it with a different name (select **File : Save as** and type the filename “`ex2.2a-dup`”). The current window is now associated with the duplicate file. Load the original file `ex2.2a` again and it will appear in a separate window (possibly on top of the first window). Move the two windows so you can see both of them.

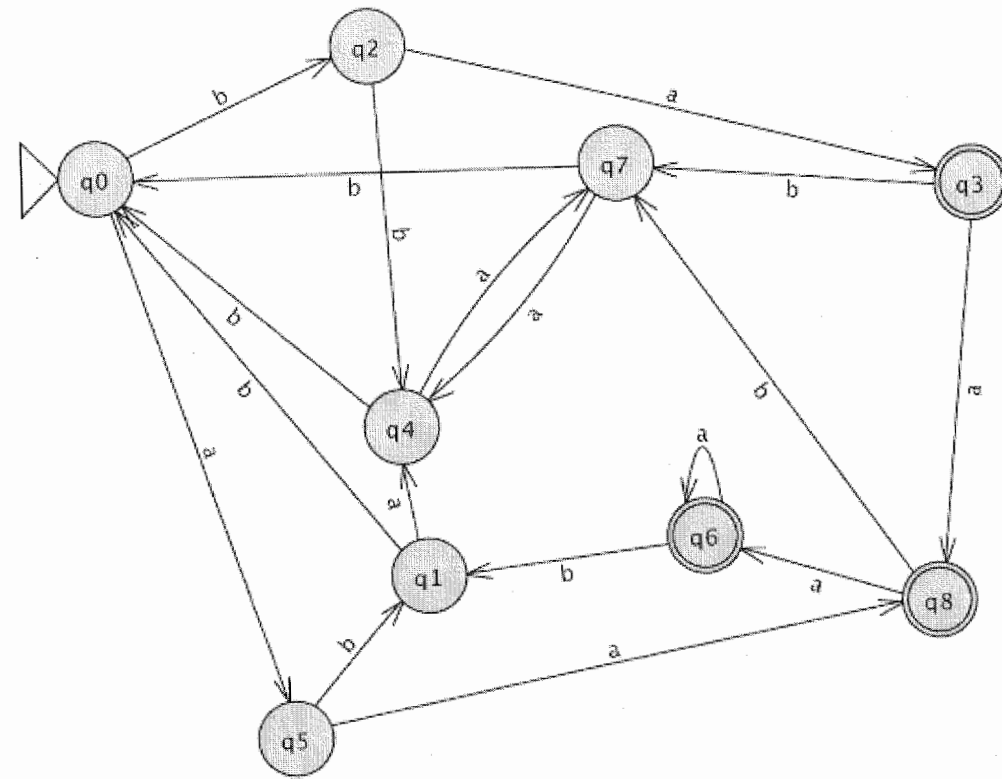


Figure 2.7: Example from file ex2.2a.

We will examine the two states  $q_0$  and  $q_1$  to see if they are distinguishable. In one of the windows, change the start state to  $q_1$ . Examine the two DFA. Are there any strings that one DFA accepts and the other DFA rejects?

We will examine several strings to see if there is any difference in acceptance and rejection. In both DFA windows, select **Input : Multiple Run**. In both windows, enter the following strings and any additional ones you'd like to try: "a", "aab", "aaaab", "baa", "baaa", and "bba". Select **Run Inputs** and examine the results. Do the strings have the same result in both DFAs? There is at least one string in which the result is **Accept** for one DFA, and **Reject** in the other DFA. Thus the two states  $q_0$  and  $q_1$  are distinguishable and cannot be combined.

Now we will examine the two states  $q_2$  and  $q_5$  to see if they are distinguishable. Dismiss the tab in both windows to go back to the DFA window. In one window change the start state to  $q_2$ , and in the other window change the start state to  $q_5$ . Select **Input : Multiple Run** again. Notice that the strings from the last run still appear in the window. Select **Run Inputs** to try these same strings. Type in additional strings and try them as well. Are these states distinguishable or indistinguishable? They are distinguishable if there is one string that accepts in one and does not accept in the other. All strings must be tested to determine if the states are indistinguishable. Clearly it is impossible to test *all* strings, so a reasonable test set should be created.

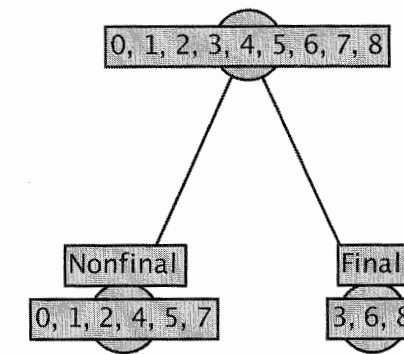


Figure 2.8: Initial split of final and nonfinal states.

### 2.2.2 Conversion Example

We go through an example of converting a DFA to a minimum state DFA. Remove the previous windows (without saving them) and load the file ex2.2a again, which should have the start state  $q_0$ . Select **Convert : Minimize DFA**. The window splits into two showing the DFA on the left and a tree of states on the right.

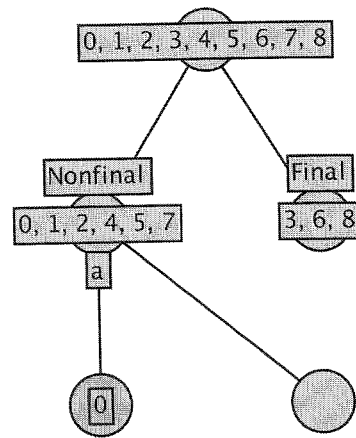
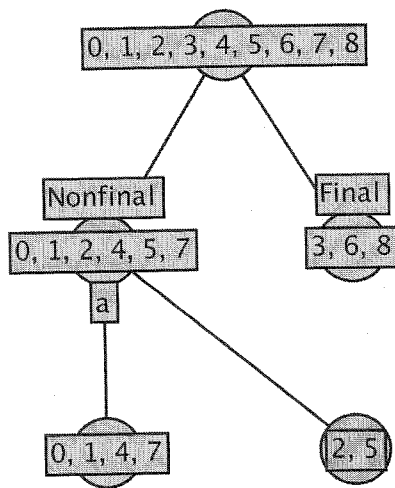
We assume that all states are indistinguishable to start with. The root of the tree contains all states. Each time we determine a distinction between states, we split a node in the tree to show this distinction. We continue to split nodes until there are no more splits possible. Each leaf in the final tree represents a group of states that are indistinguishable.

The first step in distinguishing states is to note that a final and a nonfinal state are different. The former accepts  $\lambda$  and the other does not. Thus the tree has already split the set of states into two groups of nonfinal and final states as shown in Figure 2.8.

For additional splits, a terminal will be selected that distinguishes the states in the node. If some of the states in a leaf node on that terminal go to states in one leaf node and other states on that same terminal go to states that are in another leaf node, then the node should be split into two groups of states (i.e., two new leaf nodes).

Let's first examine the leaf node of the nonfinal states (0, 1, 2, 4, 5, 7). What happens for each of these states if they process a  $b$ ? State  $q_0$  on a  $b$  goes to state  $q_2$ , state  $q_1$  on a  $b$  goes to state  $q_0$ , and so on. Each of these states on a  $b$  goes to a state already in this node. Thus,  $b$  does not distinguish these states. In JFLAP, click on the tree node containing the nonfinal states. (Click on the circle, not the label or the word Nonfinal.) The states in this node are highlighted in the DFA. Try to split this node on the terminal  $b$ . Select **Set Terminal** and enter  $b$ . A message appears informing you that  $b$  does not distinguish these states.

Again select **Set Terminal** and enter the terminal  $a$ . Since  $a$  does distinguish these states, the node is split, resulting in two new leaf nodes. The set of states from the split node must be entered into the new leaf nodes, into groups that are indistinguishable. A state number can be entered by

Figure 2.9: Split node on  $a$ .Figure 2.10: Node  $(0, 1, 2, 4, 5, 7)$  split on  $a$ .

first selecting the leaf node it will be assigned to, and then clicking on the corresponding state in the DFA. Click on the left leaf node and then click on state  $q_0$  in the DFA. The state number 0 should appear in the leaf node, as shown in Figure 2.9.

State  $q_0$  on an  $a$  goes to state  $q_5$ , which is in the node we are splitting. Note that states  $q_1$ ,  $q_4$ , and  $q_7$  on an  $a$  also go to a state in the node we are splitting. Add all of them to the same new leaf node as 0 by clicking on these states in the DFA. The remaining states,  $q_2$  and  $q_5$  on an  $a$ , go to a final state, thus distinguishing them. Click on the right new leaf node, and then click on states  $q_2$  and  $q_5$  to enter them into this node, resulting in the tree shown in Figure 2.10. To see if we have done this correctly, click on **Check Node**. Figure 2.10 shows the resulting tree after splitting this node on  $a$ .

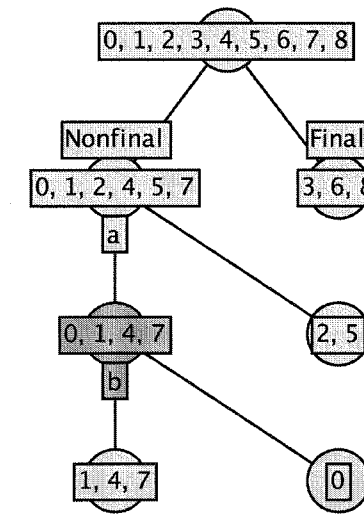


Figure 2.11: The completed tree of distinguished states.

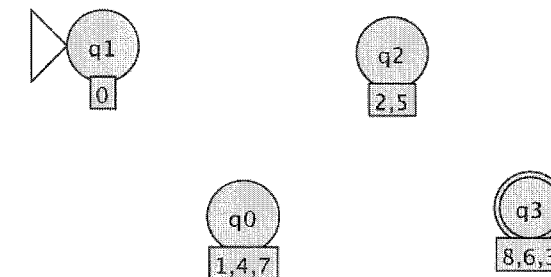


Figure 2.12: The states for the minimum DFA.

We must continually try to split nodes on terminals until there is no further splitting. Each time we split a node, we have created new groups that might now allow another group to be split that could not be split before.

Next we try to split the leaf node with states 0, 1, 4, and 7. Which terminal do you try? In this case either  $a$  or  $b$  will cause a split. We will try  $a$ . Select **Set Terminal** and enter  $a$ . Enter the split groups. State  $q_0$  on an  $a$  goes to state  $q_5$ , which is in leaf node group 2, 5, and states  $q_1$ ,  $q_4$ , and  $q_7$  on an  $a$  go to states in the leaf node we are splitting. Let's enter these states a different way. Select **Auto Partition** and the states will automatically be entered in as shown in Figure 2.11.

When the tree is complete (as it is now, convince yourself that none of the leaf nodes can be further split), then the only option visible is **Finish**. Select **Finish** and the right side of the window is replaced by the new states for the minimum DFA. There is one state for each leaf node from the tree (note the labels on the states correspond to the states from the original DFA), as shown in Figure 2.12. You may want to rearrange the states using the **Attribute Editor**.

Now add in the missing arcs in the new DFA using the **Transition Creator** tool. In the original DFA there is an  $a$  from state  $q_0$  to state  $q_5$ , so in the new DFA a transition is added

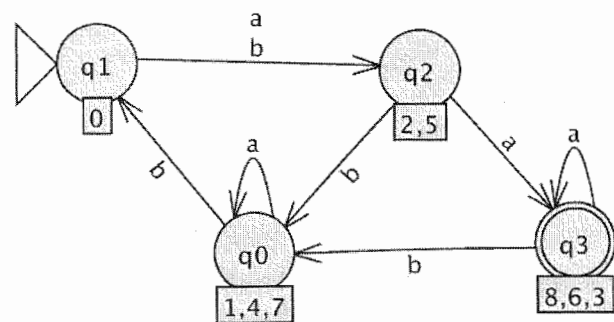


Figure 2.13: The minimum DFA.

from state  $q_1$  (representing the old state  $q_0$ ) to state  $q_2$  (representing the old state  $q_5$ ). Selecting **Hint** will add one transition for you and selecting **Complete** will complete the DFA, as shown in Figure 2.13. Selecting **Done?** will export the new DFA to its own window.

The minimum state DFA should be equivalent to the original DFA. Test this using the **Test : Compare Equivalence** option.

**Note** When you select a node and select **Set Terminal**, the node you select is split and two children appear. It is possible that the node to be split might need more children; that is, there may be 3 or more distinguished groups split on this terminal. In that case, you must add the additional leaf nodes by selecting the **Add Child** option for each additional child desired.

### 2.2.3 Algorithm

We describe the algorithm to convert a DFA  $M$  to a minimal state DFA  $M'$ .

1. Create the tree of distinguished states as follows:
  - (a) The root of the tree contains all states from  $M$
  - (b) If there are both final and nonfinal states in  $M$ , create two children of the root—one containing all the *nonfinal* states from  $M$  and one containing all the *final* states from  $M$ .
  - (c) For each leaf node  $N$  and terminal  $x$ , do the following until no node can be split:
    - i. If states in  $N$  on  $x$  go to states in  $k$  different leaf nodes,  $k > 1$ , then create  $k$  children for node  $N$  and spread the states from  $N$  into the  $k$  nodes in indistinguishable groups.
2. Create the new DFA as follows:
  - (a) Each leaf node in the tree represents a state in the DFA  $M'$  with a label equal to the states from  $M$  in the node. The start state in  $M'$  is the state that contains the start

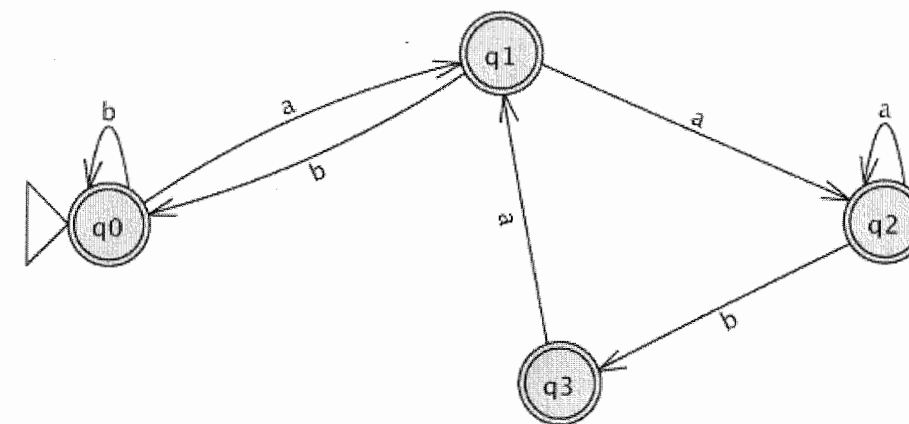


Figure 2.14: DFA from file ex2.3a.

state from  $M$  in its label. A state in  $M'$  is a final state if it contains a final state from  $M$  in its label.

- (b) For each arc in  $M$  from states  $p$  to  $q$ , add an arc in  $M'$  from the state that has  $p$  in its label to the state that has  $q$  in its label. Do not add any duplicate arcs.

### 2.3 Exercises

1. Convert the NFAs in the given files into DFAs.
  - (a) ex2-nfa2dfa-a
  - (b) ex2-nfa2dfa-b
  - (c) ex2-nfa2dfa-c
  - (d) ex2-nfa2dfa-d
  - (e) ex2-nfa2dfa-e
  - (f) ex2-nfa2dfa-f
2. Consider the language  $L = \{w \in \Sigma^* \mid w \text{ does not have the substring } abb\}$ ,  $\Sigma = \{a, b\}$ . Load the DFA in file ex2.3a shown in Figure 2.14. This DFA recognizes  $L$ . Also load the file ex2.3b. It is the NFA shown in Figure 2.15 that attempts to recognize  $L$ , but fails. Give an input string that shows why this NFA is not equivalent to this DFA.

- (b) We now want to count the number of ways to generate some of these strings; the brute-force parser will find only one, but we convert the grammar to an equivalent FA and use the fast simulator. Convert the right-linear grammar to an FA. Let  $a^\ell$  be the string of  $a$ 's of length  $\ell$ . Run the fast simulator described in Section 1.2.2 on the strings  $a^0 = \lambda$ ,  $a^1 = a$ , ..., through  $a^6 = aaaaaa$ , and count the number of ways the fast simulator finds to accept each of these strings. Remember, you can keep pressing **Keep Looking** until the final summary message appears saying how many accepting configurations it found.
- (c) Let  $A_n$  be the number of ways the FA can accept (equivalently, that the grammar can generate) the string  $a^n$ . We have  $A_0, A_1, \dots, A_6$ . Present a recursive formula for  $A_n$ , that is, determine a formula for  $A_n$  in terms of values of  $A_i$ , where  $i < n$ . *Hint: Use a counting argument. If we use the production  $S \rightarrow aaS$ , how many ways are there to generate the rest of the string without the  $aa$ ?*
- (d) Load the right-linear grammar in the file `ex3.6b`. Let  $B_n$  be the number of ways to generate  $a^n$  with this new grammar. Using your knowledge in determining a recursive formula for  $A_n$ , determine a recursive formula for  $B_n$ . *Hint: If you convert this to an FA, the number of accepting configurations during simulation of  $a^n$  is the same as the number of ways to generate  $a^n$ . For various  $a^n$ , do a fast simulation as described in Section 1.2.2 to count accepting configurations. You can manually find specific  $B_n$  this way until you see the pattern.*
8. Consider the conversion of a right-linear grammar to an FA. Sometimes the conversion of a right-linear grammar will result in a DFA, and sometimes it will result in an NFA, depending on the structure of the grammar. In this problem we explore theoretical properties of JFLAP's converter of right-linear grammars to FAs.
- (a) In Chapter 1 we explored a DFA that accepted the language over  $\Sigma = \{a, b\}$  of any number of  $a$ 's followed by any odd number of  $b$ 's. Can you create a right-linear grammar that generates this language *and* converts directly to a DFA? If you can, create the grammar with JFLAP and convert it to a DFA.
- (b) Consider a second language, the language over  $\Sigma = \{a, b, c\}$  of any number of  $a$ 's followed by any odd number of  $b$ 's, and finally suffixed with a single  $c$ . Can you create a right-linear grammar that generates this language *and* converts directly to a DFA? If you can, create the grammar with JFLAP and convert it to a DFA.
- (c) What is the general characteristic of a language for which one may construct a right-linear grammar that converts directly to a DFA? *Hint: The string  $aabb$  is in the first language. Does any other string in that language start with  $aabb$ ? The string  $aabbbc$  is in the second language. Does any other string in the language start with  $aabbbc$ ?*

## Chapter 4

# Regular Expressions

In this chapter we introduce a third type of representation of regular languages: regular expressions (REs). We describe how to edit REs, convert an RE to an equivalent NFA, and convert an FA to an equivalent RE, and then give JFLAP's formal definition of an RE.

### 4.1 Regular Expression Editing

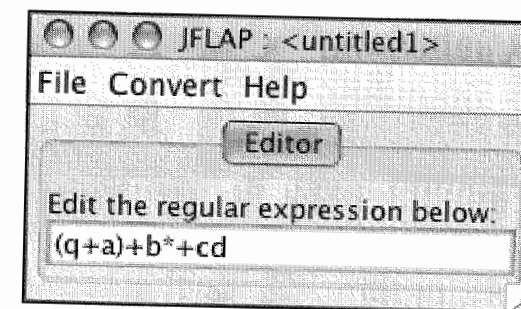


Figure 4.1: The editor for REs where the RE  $(q+a) \dots + b^* + cd$  has been entered.

In this section we learn how to edit REs. Start JFLAP; if it is already running, choose to create a new structure via the menu item **File : New**. Select **Regular Expression** from the list of new structure choices. A window will appear that is similar to Figure 4.1. Since an RE is essentially a string, JFLAP's RE editor consists of a small text field in the middle of the window.

JFLAP's REs use three basic operators. To clarify, these are not operators in the JFLAP sense, but rather the mathematical sense (e.g., pluses and minuses). The three operators in order of decreasing precedence are: the Kleene star (represented by the asterisk character  $*$ ), the concatenation operator (implicit by making two expressions adjacent), and the union operator (also called the "or" operator, represented by the plus sign  $+$ ). You may use parentheses to specify the order



of operations. Lastly, the exclamation point (!) designates the empty string, and is an easy way to enter  $\lambda$ .

A few examples of REs will help clarify JFLAP's operators' precedence. The expression  $a+b+cd$  describes the language  $\{a, b, cd\}$ , whereas  $abcd$  describes the singleton language  $\{abcd\}$ . The expression  $a(b+c)d$  describes the language  $\{abd, acd\}$ , whereas  $ab+cd$  describes the language  $\{ab, cd\}$ . The expression  $abc^*$  describes the language  $\{ab, abc, abcc, abccc, \dots\}$ , whereas  $(abc)^*$  describes the language  $\{\lambda, abc, abcabc, abcabcabc, \dots\}$ . The expression  $a+b^*$  describes the language  $\{a, \lambda, b, bb, bbb, \dots\}$ , whereas  $(a+b)^*$  describes the language  $\{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$ . The expression  $(!+a)bc$  describes the language  $\{bc, abc\}$ ; recall that ! is the user's way of entering  $\lambda$ .

In this chapter we restrict ourselves to languages over lowercase letters, but JFLAP allows any character except  $*$ ,  $+$ ,  $($ ,  $)$ , or  $!$  as part of an RE's language. Specifically, beware that the space key is a perfectly legal character for a language. For example,  $a *$  where a space follows the  $a$  (so  $a$  is followed by any number of spaces) is distinct from  $a^*$  (any number of  $a$ 's). *Note that none of the regular expressions in this chapter or its exercises have spaces in them, so do not type them in.*

We are going to enter the RE  $(q+a)+b^*+cd$ , a very simple RE that indicates that we want a string consisting of either  $q$  or  $a$ , or of any number of  $b$ 's, or the string  $cd$ . Type this RE into the text field.

## 4.2 Convert a Regular Expression to an NFA

Since REs are equivalent in power to FAs, we may convert between the two. In this section we will illustrate the conversion of an RE to an NFA. For this example we use the RE defined in Figure 4.1, the expression  $(q+a)+b^*+cd$ , also stored in file `ex4.1a`. In the window with the RE, select the menu item **Convert : Convert to NFA** to start the converter.

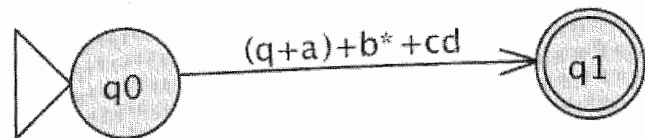



Figure 4.2: The starting GTG in the conversion.

For the purpose of the converter, we use a *generalized transition graph* (GTG), an extension of the NFA that allows *expression transitions*, transitions that contain REs. In a GTG, a configuration may proceed on a transition on a regular expression  $R$  if its unread input starts with a string  $s \in R$ ; this configuration leads to another configuration with the input  $s$  read. We start with a GTG of two states, and a single expression transition with our regular expression from the initial to the final state. The idea of the converter is that we replace each transition with new states connected by transitions on the operands of that expression's top-level operator. (Intuitively, the *top-level*

*operator* is the operator in an expression that must be evaluated last. For example, in  $ab+c$ , the top-level operator is  $+$  since the concatenation operator has higher priority and will be evaluated before the  $+$ .) We then connect these operands with  $\lambda$ -transitions to duplicate the functionality of the lost operator. In this way, at each step we maintain a GTG equivalent to the original RE. Eventually all operators are removed and we are left with single character and  $\lambda$ -transitions, at which point the GTG can be considered a proper NFA.

**Tip** You may use the Attribute Editor tool  at any point to move states around. In addition to moving states manually, with this tool the automatic graph layout algorithm may be applied, as described in Section 1.1.5.

### 4.2.1 "De-oring" an Expression Transition

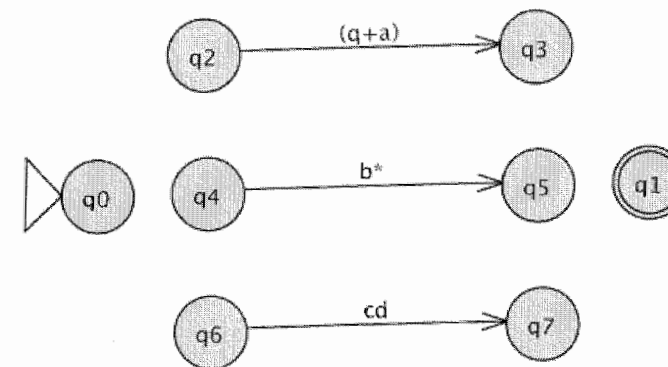





Figure 4.3: The GTG after "de-expressionifying" the first transition, but before we add the supporting  $\lambda$ -transitions.

To start converting, select the De-expressionify Transition tool . With this tool active, click on the  $(q+a)+b^*+cd$  transition. The GTG will be reformed as shown in Figure 4.3. Note that the transition has been broken up according to the top-level  $+$  union operator, and that the operands that were being "ored" have now received their own transitions. The De-expressionify Transition tool  determines the top-level operator for an expression, and then puts the operands of that operator into new expression transitions.

Note the labels near the top of the converter view: **De-oring  $(q+a)+b^*+cd$ , and 6 more  $\lambda$ -transitions needed.** These labels give an idea of what you must do next.

In this case, you must produce six  $\lambda$ -transitions so that these new six states ( $q_2$  through  $q_7$ ) and their associated transitions act like the  $+$  union operator that we have lost. To add these transitions, select the Transition Creator tool . To approximate the union functionality, you must add six  $\lambda$ -transitions, three from  $q_0$  to  $q_2$ ,  $q_4$ , and  $q_6$ , and three more to  $q_1$  from  $q_3$ ,  $q_5$ , and  $q_7$ . Intuitively,

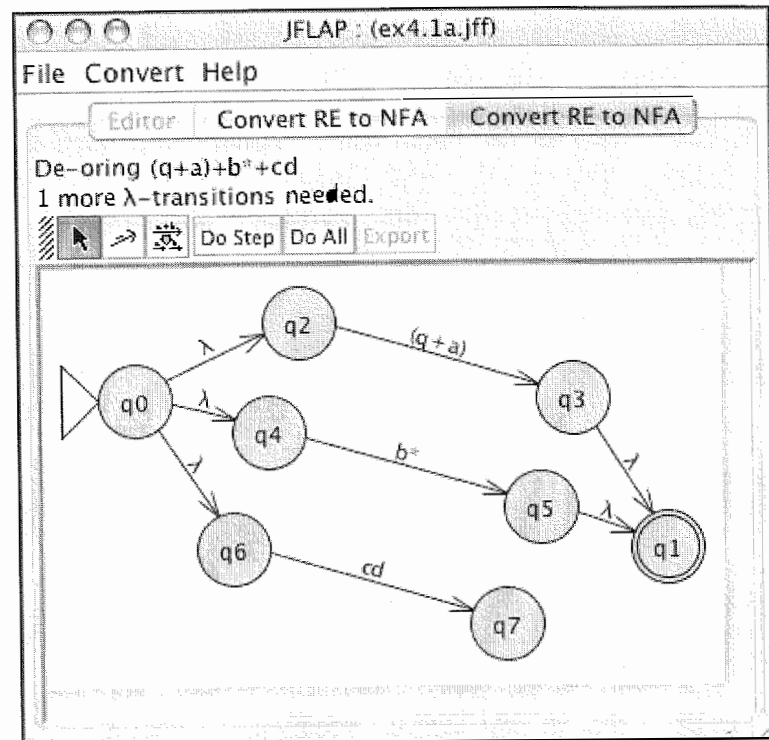


Figure 4.4: The converter window in the midst of “de-oring” the first transition. All the  $\lambda$ -transitions for this de-oring have been added, except the transition from  $q_7$  to  $q_1$ .

in going from  $q_0$  to  $q_1$ , a simulation may take the path through the  $(q+a)$  expression transition *or* the  $b^*$  expression transition *or* the  $cd$  expression transition. In short, these  $\lambda$ -transitions help to approximate the functionality of the lost  $+$  operator on these operands. Use the Transition Creator tool  $\rightarrow$  to create these. All transitions are  $\lambda$ -transitions, so JFLAP does not bother asking for labels. As you add transitions, the label at the top of the window decrements the number of transitions needed. Figure 4.4 shows an intermediate point in adding these transitions, with only the transition from  $q_7$  to  $q_1$  not created. When you finish adding these transitions to the GTG, JFLAP allows you to “de-expressionify” another transition.

#### 4.2.2 “De-concatenating” an Expression Transition

Once you finish “de-oring” the first transition, you have three expression transitions. We will reduce  $cd$  next; the top-level operator for this expression is the concatenation operator. Select the De-expressionify Transition tool  $\rightarrow$  once more, and click on the  $cd$  transition. In Figure 4.5 you see the result. Note that JFLAP informs us that we are **De-concatenating  $cd$**  and that we have **3 more  $\lambda$ -transitions needed**; similar to de-oring, de-concatenating requires the addition of  $\lambda$ -transitions to approximate the lost concatenation operator.

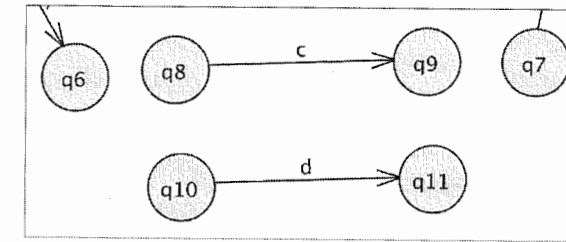


Figure 4.5: The beginning of de-concatenating the expression transition  $cd$ . States and transitions extraneous to the de-concatenating are cropped out.

We require three  $\lambda$ -transitions: one from  $q_6$  to  $q_8$ , another from  $q_9$  to  $q_{10}$ , and a last one from  $q_{11}$  to  $q_7$ . Configurations on  $q_6$  will have to satisfy the  $c$  expression (between  $q_8$  and  $q_9$ ), and then satisfy the  $d$  expression (between  $q_{10}$  and  $q_{11}$ ) before proceeding to  $q_7$ . This arrangement is functionally equivalent to  $c$  concatenated with  $d$ .

#### A remedy of errors

Select the Transition Creator tool  $\rightarrow$ . Instead of adding the right transitions, let’s add an incorrect transition! Create a transition from  $q_6$  to  $q_{10}$ . With this transition, the configuration can proceed from  $q_6$  to the  $d$  portion, bypassing  $c$ . This is incorrect. A dialog box will report **A transition there is invalid**, and the transition will not be added.

Although checking for wrong transitions is universal to the converter no matter what operator you are splitting on, the de-concatenating process has some additional restrictions. Add a transition from  $q_{11}$  to  $q_7$ . This is perfectly valid! However, JFLAP reports in a dialog, **That may be correct, but the transitions must be connected in order**. In this case, this means you must first connect  $q_6$  to  $q_8$ , and then  $q_9$  to  $q_{10}$ , and only then may you connect  $q_{11}$  to  $q_7$ . Add these transitions now.

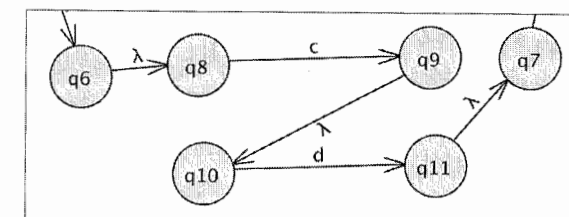


Figure 4.6: The finished de-concatenating of the expression transition  $cd$ .

The relevant portion of the automaton will resemble Figure 4.6. Since you have finished the deconcatenation of  $cd$ , you may now reduce another expression transition. Select the De-expressionify Transition tool  $\overline{\text{X}}$  again. Recall that the converter recursively breaks down expression transitions until they are either one character or  $\lambda$ -transitions. If you click on the  $c$  transition, the message **That's as good as it gets** appears to inform you that you needn't reduce that transition.

#### 4.2.3 "De-staring" a Transition

We will reduce the  $b^*$  transition next. With the De-expressionify Transition tool  $\overline{\text{X}}$  active, click the  $b^*$  transition. Kleene stars may have only one operand, in this case  $b$ . As we see in Figure 4.7, the  $b$  has been separated into a new portion of the automaton. JFLAP tells us that we are **De-staring  $b^*$**  and that there are **4 more  $\lambda$ -transitions needed**.

Similar to concatenations and ors, we must add  $\lambda$ -transitions to duplicate the functionality of the Kleene star. The four transitions that JFLAP wants are from  $q_4$  to  $q_{12}$  and  $q_{13}$  to  $q_5$  (to allow configurations to read a  $b$  from their input), and another from  $q_4$  to  $q_5$  (to allow zero  $b$ 's to be read), and the last from  $q_5$  to  $q_4$  (to allow for repeat reading of  $b$ ). Select the Transition Creator tool  $\overrightarrow{\text{X}}$ , and add these transitions so the relevant portion of the GTG resembles Figure 4.8.

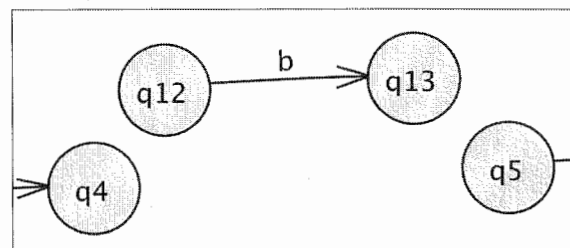


Figure 4.7: The beginning of de-staring the expression transition  $b^*$ . States and transitions extraneous to the de-staring are cropped out.

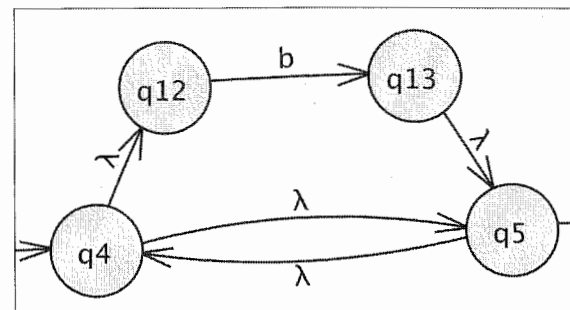


Figure 4.8: The finished de-staring of the expression transition  $b^*$ .

#### 4.2.4 Surrounding Parentheses

The only remaining existing transition incompatible with an NFA is the  $(q+a)$  transition, which has surrounding parentheses. The parentheses are the top-level operator since they indicate that their contents must be evaluated first, and only when that evaluation finishes do the parentheses finish evaluating. However, when the parentheses surround the entire expression, they are completely unnecessary. Activate the De-expressionify Transition tool  $\overline{\text{X}}$ , and click on the  $(q+a)$  transition. The surrounding parentheses will disappear, leaving you with  $q+a$ . No  $\lambda$ -transitions are needed.

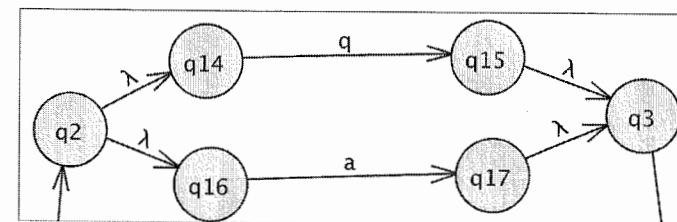


Figure 4.9: The finished de-oring of the expression transition  $q+a$ .

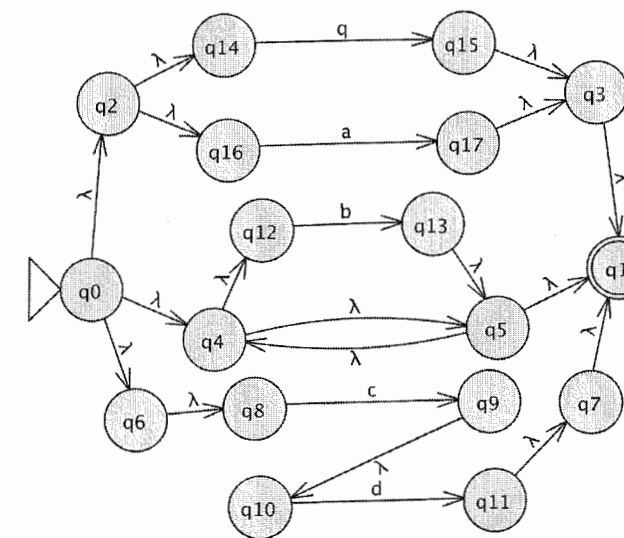


Figure 4.10: The NFA that recognizes the language  $(q+a)+b^*+cd$ .

To finish, use the De-expressionify Transition tool  $\overline{\text{X}}$  tool once more to break  $q+a$  by the  $+$  operator. Connect  $\lambda$ -transitions similar to the procedure described in Section 4.2.1, so that the

relevant section of the GTG resembles Figure 4.9, and overall the automaton resembles Figure 4.10. The GTG is now a proper NFA, so the conversion to an NFA is finished! You may press the **Export** button to put the automaton in a new window.

#### 4.2.5 Automatic Conversion

Dismiss the **Convert RE to NFA** tab now. Once you have returned to the RE editor, select the menu item **Convert : Convert to NFA**. We shall convert the same RE again, but we'll do it automatically this time!

Once you see the converter view with the GTG as pictured in Figure 4.2, press **Do Step**. A step in this conversion is the reduction of a single expression transition. There is only one expression transition, the  $(q+a)+b^*+cd$  transition, so that is reduced and the requisite  $\lambda$ -transitions are added without intervention from the user.

The second option is **Do All**; this is functionally equivalent to pressing **Do Step** until the conversion finishes. This is useful if you want the equivalent NFA immediately. Press **Do All**; the finished NFA will appear in the window, ready to be exported.

#### 4.2.6 Algorithm to Convert an RE to an NFA

1. Start with an RE  $R$ .
2. Create a GTG  $G$  with a single initial state  $q_0$ , single final state  $q_1$ , and a single transition from  $q_0$  to  $q_1$  on the expression  $R$ .
3. Although there exists some transition  $t \in G$  from states  $q_i$  to  $q_j$  on the expression  $S$  longer than one character, let  $\phi$  be the top-level operator of the expression  $S$ , and let  $[\alpha_1, \alpha_2, \dots, \alpha_\psi]$  be the ordered list of operands of the operator  $\phi$  (since parenthetical and Kleene star operators take exactly one operand  $\psi = 1$  in those cases).
  - (a) If  $\phi$  is a parenthetical operator, replace  $t$  with an expression transition on  $\alpha_1$  from  $q_i$  to  $q_j$ .
  - (b) If  $\phi$  is a Kleene star operator ( $*$ ), create two new states  $q_x$  and  $q_y$  for  $G$ , remove  $t$ , and create an expression transition on  $\alpha_1$  from  $q_x$  to  $q_y$ , and create four  $\lambda$ -transitions from  $q_i$  to  $q_x$ ,  $q_y$  to  $q_j$ ,  $q_i$  to  $q_j$ , and  $q_j$  to  $q_i$ .
  - (c) If  $\phi$  is a union operator ( $+$ ), remove  $t$ , and for each  $k$  from 1 through  $\psi$  (i) create two new states  $q_{x_k}$  and  $q_{y_k}$ , (ii) create an expression transition on  $\alpha_k$  from  $q_{x_k}$  to  $q_{y_k}$ , and (iii) create two  $\lambda$ -transitions, from  $q_i$  to  $q_{x_k}$  and from  $q_{y_k}$  to  $q_j$ .
  - (d) If  $\phi$  is a concatenation operator, remove  $t$ , and for each  $k$  from 1 through  $\psi$  (i) create two new states  $q_{x_k}$  and  $q_{y_k}$ , (ii) create an expression transition on  $\alpha_k$  from  $q_{x_k}$  to  $q_{y_k}$ , and

- (iii) if  $k > 1$  create a  $\lambda$ -transition from  $q_{y_{k-1}}$  to  $q_{x_k}$ . Finally, create two  $\lambda$ -transitions, one from  $q_i$  to  $q_{x_1}$  and another from  $q_{y_\psi}$  to  $q_j$ .

4. The GTG is now a proper NFA. The conversion is finished.

### 4.3 Convert an FA to a Regular Expression

The conversion of an FA to an RE follows logic that is in some respects reminiscent of the RE to NFA conversion described in Section 4.2. We start with an FA that we consider a GTG for the purposes of conversion. We then remove states successively, generating equivalent GTGs until only a single initial and single final state remain. JFLAP then uses a formula to express the simplified GTG as a regular expression.

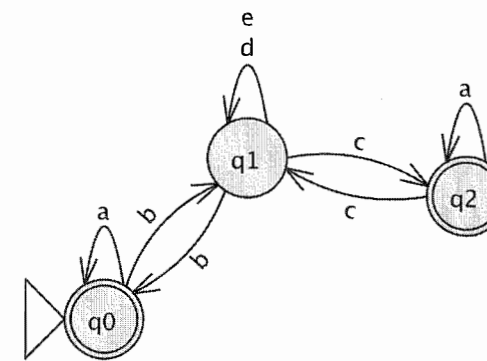



Figure 4.11: The FA we convert to an RE.

In this walk-through we convert the automata pictured in Figure 4.11 to a regular expression. This automata is stored in the file **ex4.3a**. Open this automata. Choose the menu item **Convert : Convert FA to RE** to begin converting. Your window will resemble Figure 4.12.

#### 4.3.1 Reforming the FA to a GTG

The algorithm to convert an FA to an RE requires first that the FA be reformed into a GTG with a single final state, an initial state that is not a final state, and exactly one transition from  $q_i$  to  $q_j$  for every pair of states  $q_i$  and  $q_j$  ( $i$  may equal  $j$ ).

##### Reform FA to have a single noninitial final state

There are two things wrong with our FA's final states: there are two final states, and one of them is also the initial state. We must reform the automaton so that it has exactly one final state and ensure that that final state is not the initial state. To do this JFLAP first requires that a new state be created: select the State Creator tool , and click somewhere on the canvas to create a new state. (Similar to the conversion from an RE to an NFA, this converter also displays directions above the editor. At this stage it tells you **Create a new state to make a single final state.**)

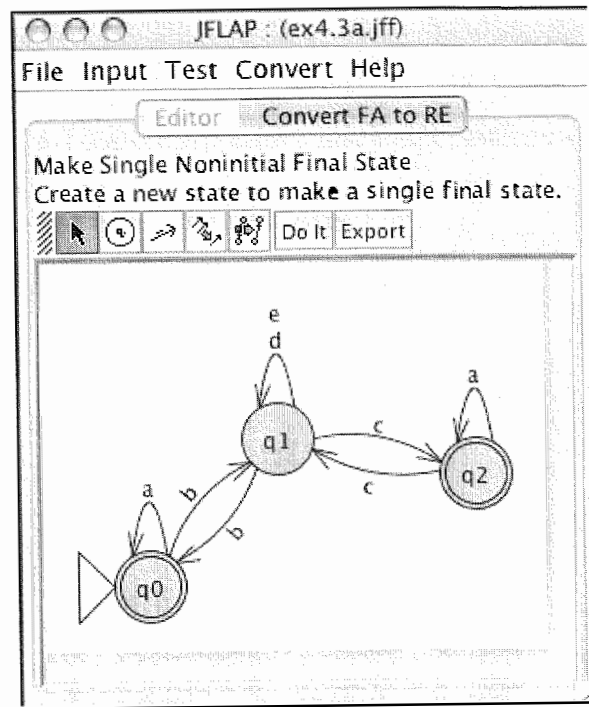


Figure 4.12: The starting window when converting an FA to an RE.

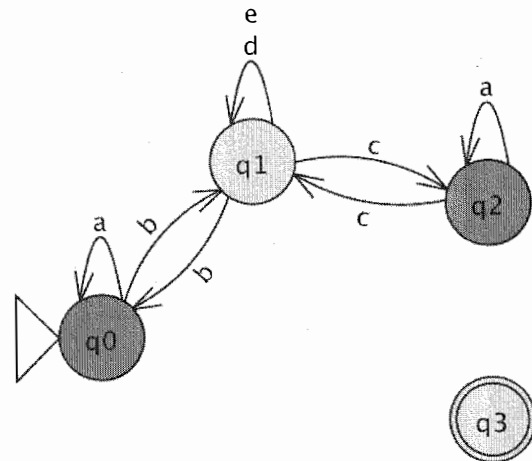



Figure 4.13: The FA after a new final state is created.

Once this new state is created, the FA will resemble Figure 4.13. Note that this new state is the final state, and those states that were previously final states are now regular states and have been highlighted. JFLAP directs you to **put  $\lambda$ -transitions from old final states to new**. Select the Transition Creator tool  and create transitions from each of the highlighted states to the new final states. JFLAP assumes that every transition is a  $\lambda$ -transition and does not query for the

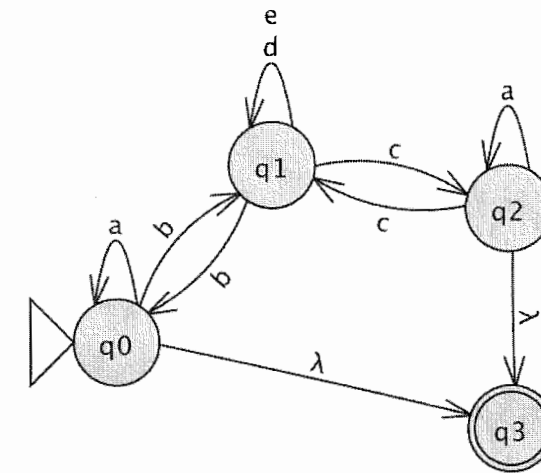




Figure 4.14: The FA after the  $\lambda$ -transitions have been made from the old final states to the new final state.

transition label. As you create each  $\lambda$ -transition, the source state will be de-highlighted. When you finish, your FA will resemble Figure 4.14.

#### Collapse multiple transitions

One of the requirements of this algorithm is that for every pair of states  $q_i$  and  $q_j$  there must be exactly one transition from  $q_i$  to  $q_j$ . Half of this requirement is that there cannot be more than one transition from  $q_i$  to  $q_j$ . Consider the two loop transitions for  $q_1$  on  $d$  and  $e$ . We can satisfy the requirement by replacing these two transitions with the single expression transition  $d+e$ , which indicates that we may proceed on either  $d$  or  $e$ .

Select the Transition Collapser tool , and click on either the  $d$  or  $e$ . When you click on a transition that goes from  $q_i$  to  $q_j$ , this tool reforms all transitions from  $q_i$  to  $q_j$  into a single transition where the labels of the removed transitions are separated by  $+$  operators. The new transition will be either  $d+e$  or  $e+d$ , either of these is equivalent, of course, but for the sake of this discussion's simplicity we assume the result was  $d+e$ . With this step, our GTG is no longer a proper FA. The GTG is shown in Figure 4.15.

In general, if more than one pair of states have more than one transition, use the Transition Collapser tool  on their transitions as well.

#### Add empty transitions

Recall once more that every pair of states  $q_i$  and  $q_j$  must have exactly one transition from  $q_i$  to  $q_j$ . This means that if no transition exists, an *empty transition* (on the empty set symbol  $\emptyset$ ) must

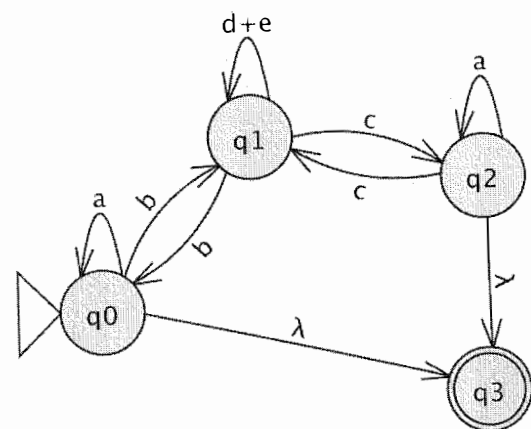



Figure 4.15: The GTG after the  $d$  and  $e$  loop transitions on  $q_1$  are combined into  $d+e$ .

be created! Select the Transition Creator tool  again, and create a transition from  $q_0$  to  $q_2$ . A transition on  $\emptyset$  will appear.

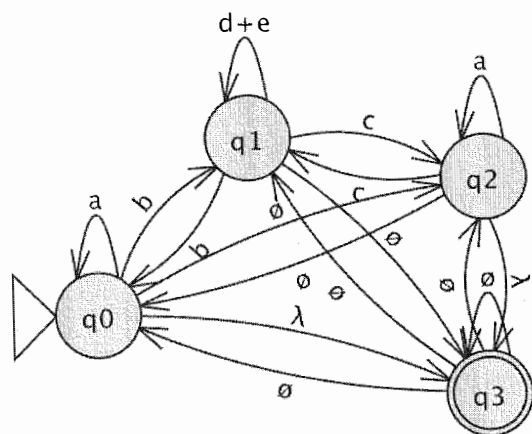


Figure 4.16: The FA after the addition of empty transitions.

The essential distinction between GTGs and FAs is that FA transitions describe a single string, while GTG transitions describes sets of strings. In this particular case, we are creating transitions on the empty set of strings, hence transitions on  $\emptyset$ . Similar to the earlier creation of  $\lambda$ -transitions, JFLAP assumes you are creating empty transitions. As you proceed, JFLAP will inform you how many more empty transitions are required. Seven are required in all: from  $q_0$  to  $q_2$ ,  $q_1$  to  $q_3$ ,  $q_2$  to  $q_0$ ,  $q_3$  to  $q_0$ ,  $q_3$  to  $q_1$ ,  $q_3$  to  $q_2$ , and a loop transition on  $q_3$  ( $q_3$  to  $q_3$ ). When you finish, your GTG will resemble Figure 4.16.


### 4.3.2 Collapse Nonfinal, Noninitial States

Now we have a GTG with a single final state, an initial state that is not a final state, and for every pair of states  $q_i$  and  $q_j$  there is exactly one transition from  $q_i$  to  $q_j$ . The next step is to iteratively remove every state in the GTG except the final state and the initial state. As each state is removed, we adjust the transitions remaining to ensure the GTG after the state removal is equivalent to the GTG before the state removal.

Transitions		
Select to see what transitions were co...		
From	To	Label
0	0	a
0	1	b
0	3	$\lambda$
1	0	b
1	1	$(e+d)+ca^*c$
1	3	$ca^*$
3	0	$\emptyset$
3	1	$\emptyset$
3	3	$\emptyset$

Finalize

Figure 4.17: The window that shows the replacement transitions when removing a state.

The states can be collapsed in any order. However, to understand the following discussion, you will need to collapse states in the given order. Select the State Collapser tool . Once selected, click first on state  $q_2$ . A window like the one shown in Figure 4.17 appears that informs you of the new labels for transitions before the collapse occurs. Let  $r_{ij}$  be the expression of the transition from  $q_i$  to  $q_j$ . The rule is, if we are removing  $q_k$ , for all states  $q_i$  and  $q_j$  so that  $i \neq k$  and  $j \neq k$ ,  $r_{ij}$  is replaced with  $r_{ij} + r_{ik}r_{kk}^*r_{kj}$ . In other words, we compensate for the removal of  $q_k$  by encapsulating in the walk from  $q_i$  to  $q_j$  the effect of going from  $q_i$  to  $q_k$  (hence  $r_{ik}$ ), then looping on  $q_k$  as much as we please (hence  $r_{kk}^*$ ), and then proceeding from  $q_k$  to  $q_j$  (hence  $r_{kj}$ ). Lastly, note that  $\emptyset$  obeys the following relations: if  $r$  is any regular expression,  $r + \emptyset = r$ ,  $r\emptyset = \emptyset$ , and  $\emptyset^* = \lambda$ .

Select the row that describes the new transition from  $q_1$  to  $q_1$  (the loop transition on  $q_1$ ),  $d+e+ca^*c$ . The transitions from which this new transition is composed are highlighted in the GTG. There are two paths that must be combined into one expression transition, the walk from  $q_1$  to  $q_1$ ,  $d+e$ , and the alternative walk from  $q_1$  to  $q_1$  that goes through  $q_2$ ,  $ca^*c$ . More formally,  $r_{1,1} = d+e$ ,  $r_{1,2} = r_{2,1} = c$ , and  $r_{2,2} = a$ , so the new transition is  $r_{1,1} + r_{1,2}r_{2,2}^*r_{2,1} = d+e+ca^*c$  as JFLAP

indicates.

The rules for operations on the empty set are more unfamiliar. Select the row that describes the new transition from  $q_0$  to  $q_1$ . There are two paths that must be combined into one expression transition, the walk from  $q_0$  to  $q_0$ ,  $b$ , and the alternative walk from  $q_0$  to  $q_1$  that goes through  $q_2$ ,  $\emptyset a^* c$ . More formally,  $r_{0,1} = b$ ,  $r_{0,2} = \emptyset$ ,  $r_{2,2} = a$ , and  $r_{2,1} = c$ , so the new transition is  $r_{0,1} + r_{0,2} r_{2,2}^* r_{2,1} = b + \emptyset a^* c$ . The concatenation of any expression with the empty set is the empty set, so  $\emptyset a^* c = \emptyset$ , so  $b + \emptyset a^* c = b + \emptyset$ . The union of the empty set with any expression is that expression, so  $b + \emptyset = b$ , which is the new expression from  $q_0$  to  $q_1$ .

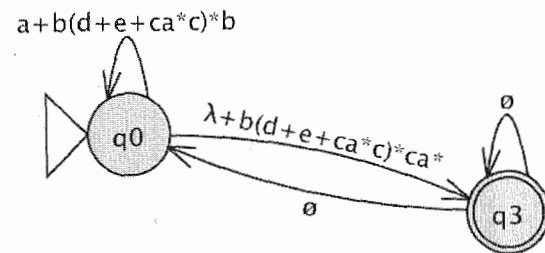


Figure 4.18: The finished GTG after the removal of  $q_2$  and  $q_1$ .

Inspect all the other replacements to see if you can figure out the formula, and then reduce it to the label shown in Figure 4.17. Then click **Finalize**. The transitions listed will be replaced, and  $q_2$  will be removed. Repeat this process with  $q_1$ . Note there are only four replacements, and some of the labels are quite long. (You might have to resize the window to see the complete labels.) When  $q_1$  is removed, your GTG will resemble Figure 4.18.

### 4.3.3 Regular Expression Formula

At this point your GTG should have two states—one final and one initial—and resemble Figure 4.18. Let  $r_{xy}$  be the expression of the transition from  $q_x$  to  $q_y$ . For a GTG in this form, where  $q_i$  is the initial state and  $q_j$  is the final state, the equivalent RE  $r$  is given by equation 4.1.

$$r = (r_{ii}^* r_{ij} r_{jj}^* r_{ji})^* r_{ii}^* r_{ij} r_{jj}^* \quad (4.1)$$

The conversion is now finished, and JFLAP displays the RE of equation 4.2, derived from equation 4.1.

$$(a+b(d+e+ca^*c)^*b)^*(\lambda+b(d+e+ca^*c)^*ca^*) \quad (4.2)$$

At this point, you may press **Export** to put the finished RE in a new window.

**Note** If for your input FA any of these steps are unnecessary, JFLAP will skip over them. In the extreme case, if you have an FA with two states (one initial, the other final), and with four transitions (a loop on the initial, a loop on the final, another from the initial to final, and a last one from the final to the initial), JFLAP will skip everything and display the finished RE!

### 4.3.4 Algorithm to Convert an FA to an RE

1. Start with an FA, though we consider it a GTG  $G$  for the purpose of the algorithm.
2. Let  $F$  be the set of  $G$ 's final states, and  $q_0$  be its initial state. If  $|F| > 1$  or  $F = \{q_0\}$ , create a new state  $q_f$ , produce  $\lambda$ -transitions for every  $q_i \in F$  from  $q_i$  to  $q_f$ , and make  $q_f$  the only final state.
3. Let  $S$  be the set of  $G$ 's states. For every  $(q_i, q_j) \in S \times S$ , let  $L = \{\ell_1, \ell_2, \dots, \ell_n\}$  be the set of expressions of transitions from  $q_i$  to  $q_j$ . Let  $e = \emptyset$  if  $|L| = 0$  and  $e = \ell_1 + \ell_2 + \dots + \ell_n$  otherwise, and replace all transitions from  $q_i$  to  $q_j$  with a single transition from  $q_i$  to  $q_j$  on the expression  $e$ .
4. Let  $T$  be the set of  $G$ 's nonfinal, noninitial states. Let  $r_{xy}$  be the expression of the transition from  $q_x$  to  $q_y$ . For every  $q_k \in T$ , for every  $(q_i, q_j) \in (T - \{q_k\}) \times (T - \{q_k\})$  replace  $r_{ij}$  with  $r_{ij} + r_{ik} r_{kk}^* r_{kj}$ , and finally remove  $q_k$  from  $G$ . (Note: If  $r$  is any regular expression,  $r + \emptyset = r$ ,  $r\emptyset = \emptyset$ , and  $\emptyset^* = \lambda$ .)
5.  $G$  now has two states: the initial state  $q_0$  and single final state  $q_f$ . The equivalent regular expression is  $r = (r_{00}^* r_{0f} r_{ff}^* r_{f0})^* r_{00}^* r_{0f} r_{ff}^*$ .

## 4.4 Definition of an RE in JFLAP

Let  $\Sigma$  be some alphabet. The set of all possible regular expressions is given by  $R$ .

$$R = \{\emptyset, \lambda\} \cup \Sigma \cup \{ab \mid a, b \in R\} \cup \{a+b \mid a, b \in R\} \cup \{a^* \mid a \in R\} \cup \{(a) \mid a \in R\}$$

## 4.5 Summary

In Section 4.1 we learned how to edit regular expressions (REs). JFLAP respects the following operators in order of decreasing precedence: the Kleene star (the  $*$  character), concatenation (implicit by making two expressions adjacent), and lastly, union (the  $+$  character). Parentheses