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

![JFLAP Interface]

Figure 4.1: The editor for REs where the RE \( (q+a) \ldots + b^*+cd \) 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 \( \{ab, abc, abc^2, abc^3, \ldots\} \), whereas \( (abc)^* \) describes the language \( \{\lambda, abc, abcabc, abcabcabc, \ldots\} \). The expression \( a+b^* \) describes the language \( \{a, \lambda, b, bb, bbb, \ldots\} \), whereas \( (a+b)^* \) describes the language \( \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\} \). The expression \( (a+\lambda)bc \) 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.](image)

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
4.2. CONVERT A REGULAR EXPRESSION TO AN NFA

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

![Diagram showing transition labeled \((q+a)\) from \(q2\) to \(q3\), \(b^*\) from \(q4\) to \(q5\) and \(q1\), and \(cd\) from \(q6\) to \(q7\).]

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 \(\text{X}\). 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 \(\text{X}\) 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 \(\text{X}\). 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,
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 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 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.
4.2. CONVERT A REGULAR EXPRESSION TO AN NFA

![Diagram of NFA]

Figure 4.5: The beginning of deconcatenating the expression transition cd. States and transitions extraneous to the deconcatenating are cropped out.

We require three λ-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 \( q_8 \) 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 deconcatenating 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.

![Diagram of completed NFA]

Figure 4.6: The finished deconcatenating 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 \( \mathcal{F} \) 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 \( \mathcal{F} \) 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 \( \rightharpoonup \), and add these transitions so the relevant portion of the GTG resembles Figure 4.8.

![Figure 4.7](image1)

**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](image2)

**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 \(\mathcal{X}\), 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 \(\mathcal{X}\) 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 \((q + a)* + cd\) 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 \([\alpha_1, \alpha_2, \ldots, \alpha_\psi]\) 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
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.

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 $\varnothing$, 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 λ-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 λ-transition and does not query for the
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 $\rightarrow$ 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 Table](image)

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 Collider tool \( \mathcal{E} \). 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 \( r_{1,1} + r_{1,2} r_{2,1}^* r_{1,2} = d + e + ca^*c \) 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 $q_1$.

$$a + b(d + e + ca^*c)^*b$$

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 $q_f$ is the final state, the equivalent RE $r$ is given by equation 4.1.

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

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

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

At this point, you may press Export to put the finished RE in a new window.
### 4.4 Definition of an RE in JFLAP

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

\[
R = \{ \emptyset, \lambda \} \cup \Sigma \cup \{ab | a, b \in R\} \cup \{a+b | a, b \in R\} \cup \{a^* | a \in R\} \cup \{[a] | a \in R\}
\]

### 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
may be used to specify the order of operations. The character $!$ is used to enter the empty string. All other characters may be used as alphabet symbols, including spaces.

In Section 4.2 we learned how to use JFLAP to convert an RE to an NFA. We use the idea of a generalized transition graph (GTG), which is essentially an FA that allows regular expressions in transitions (called expression transitions). The converter starts with a two-state GTG, with a transition on the starting regular expression from the initial to final state. While there are expression transitions, JFLAP will break the transition into subcomponents that contain the operands of the top-level operator of the expression; the user is responsible for connecting the subcomponents with $\lambda$-transitions to duplicate the lost functionality of the operator. The converter finishes when all transitions are single-character or $\lambda$-transitions.

In Section 4.3 we learned how to use JFLAP to convert an FA to an RE. We start by considering the FA as a GTG, and reforming the GTG so it has a single final state and a nonfinal initial state, and so that for any two states in the GTG there is exactly one transition from the first state to the second state. We then successively remove all states but the final and initial state; with each removal JFLAP adjusts the remaining transitions to duplicate the functionality of the lost state and its transitions so that each GTG is equivalent to the original FA. Eventually, only the initial and final states remain, to which JFLAP applies a known formula to derive the final RE equivalent to the FA.

In Section 4.4 we presented JFLAP's formal definition of an RE.

### 4.6 Exercises

1. Using JFLAP, first convert the following REs to NFAs. Then run the multiple run tester and list six input strings that are in the language.

   (a) $abc+a^*$
   (b) $a^*b^*+c$
   (c) $(ab+\lambda)b$
   (d) $(a+c)a^*$
   (e) $(a+bc+d)$
   (f) $(cd+d)^*$
   (g) $(a+b)^*cd$

2. Write a regular expression for each of the following languages. Enter it in JFLAP, quickly convert it to an NFA (select Do All), and then use the multiple run window to verify the strings below. You should check additional strings as well.
4.6. EXERCISES

(a) \( \Sigma = \{a,b\} \). The language is strings with an even number of \( a \)'s followed by an odd number of \( b \)'s. For example, accepted strings include \( aab, b, aaaaab, \) and \( aaaabbbb \), and rejected strings include \( aba \) and \( aaab \).

(b) \( \Sigma = \{a,b\} \). The language is strings with an even number of \( a \)'s that have exactly one \( b \). For example, accepted strings include \( aba, b, aaaaaba, \) and \( aaaba, \) and rejected strings include \( abab \) and \( aaab \).

(c) \( \Sigma = \{a,b\} \). The language is strings with zero or more \( a \)'s followed by three or fewer \( b \)'s. For example, accepted strings include \( aab, bbb, aaaaab, \) and \( aaaa, \) and rejected strings include \( aabbbb \) and \( aabaab \).

(d) \( \Sigma = \{a,b\} \). The language is strings in which every \( a \) must have a \( b \) adjacent to it on both sides. For example, accepted strings include \( bababab, bab, bbaab, \) and \( babbbabbb, \) and rejected strings include \( abab \) and \( baab \).

(e) \( \Sigma = \{a,b\} \). The language is strings in which every pair of adjacent \( a \)'s is followed by a \( b \). For example, accepted strings include \( babab, baab, aabbaab, \) and \( aaba, \) and rejected strings include \( aaab \) and \( baab \).

(f) \( \Sigma = \{a,b\} \). The language is strings in which every \( a \) is followed by an odd number of \( b \)'s. For example, accepted strings include \( bb, bbbab, abbbab, \) and \( babbbababbb, \) and rejected strings include \( ba, abab, \) and \( bbbab. \)

(g) \( \Sigma = \{a,b\} \). The language is strings in which every group of adjacent \( b \)'s is divisible by two or three. For example, accepted strings include \( bbabb, bbbbba, abbbab, \) and \( bbbababbbba, \) and rejected strings include \( ba, abab, \) and \( bbbb. \)

3. Revisit the example in Section 4.3 with ex4.3a in which \( q_2 \) is removed first. In the process of removing \( q_2 \), how many new labels are empty transitions? Now redo the example and remove \( q_1 \) first. In the process of removing \( q_1 \), how many new labels are empty transitions?

4. Convert the following FAs to REs.

(a) ex4.nfa2re-a
(b) ex4.nfa2re-b
(c) ex4.nfa2re-c
(d) ex4.nfa2re-d
(e) ex4.nfa2re-e
(f) ex4.nfa2re-f
(g) ex4.nfa2re-g
(h) ex4.nfa2re-h
5. Load file ex4.1a and convert it to an FA. Then convert the FA back into a regular expression by first converting it to a DFA, then a minimal state DFA, and then a regular expression. The original RE and the resulting RE are different. Explain.

6. Load file ex4.3a and convert it to an RE. Then convert the resulting RE to a FA, then a DFA, and then a minimal state DFA. What do you observe about the original DFA and the resulting minimal state DFA?